XInC2 C Programming Primer

Thia Wyrod

26 January 2015

Contents

Part I

Introduction

Welcome to programming in C for XInC2! The intended audience of this guide is those interested in developing firmware for an embedded system board using a XInC2 processor. This guide is written with the assumption that the reader has at least a cursory knowledge of C, but assumes very little else. It is also the author's intent that this guide not only enable the reader to write firmware in C for XInC2, but to also enable the reader to become a more efficient programmer in general.

Finally, if one runs into any issues in attempting to install or run any part of the toolchain, one should first consult Part VI, which offers solutions for several common issues pertaining to installation.

0.1 Notation Used in the Guide

Throughout the guide, there will be references to terminal commands and paths, especially in instructions pertaining to installation and building firmware. Generally, these references will be surrounded by quotation marks (" or '); when running these commands, one should not actually type the literal quotation marks. Variable names, where one should substitute in a relevant expression in the command, will be surrounded by <>, such as <VAR_NAME>. The surrounding chars should again not be typed out.

Without further ado, let's move onto the guide!

0.2 Prerequisite Software

- CMake 2.8+
- Python 3.4+
  - Cogapp
- SXC Cross-Compilation Suite (included with this guide)
- A cross-compilation-capable build chain

0.3 Software Overview

0.3.1 CMake

CMake is a cross-platform build system and pseudo-scripting language. It enables developers to write instructions on how a project's source code should be compiled and linked at a high level. CMake then translates these high-level instructions (stored in a project's CMakeLists.txt file) into a selected low-level build system's make or project files. For example, CMake can generate

Code::Blocks project files or Unix Makefiles, which are then used to actually invoke a compiler to build software. By using CMake, a developer is no longer restricted to a specific build system to compile their software, and different collaborators working on the same codebase may use radically different IDEs as a result.

0.3.2 Python

Python is a high-level programming/scripting language combined with a run-time environment/interpreter which is used to actually execute Python code. Python, specifically the Cogapp application written in it by Ned Batchelder, is used in the XInC2 build process. It is used to generate metadata based on a project's source code, which is needed to create a binary firmware image that can be loaded onto a XInC2 board.

0.3.3 SXC Cross-Compilation Suite

The SXC Cross-Compilation Suite is a collection of software written by Eleven Engineering which is used to compile and load firmware written for XInC2. It includes C/C++ compiler and linkers, an optimizing assembler, and tools to package compiled firmware into a binary image and load it onto a development board.

0.3.4 Cross-Compilation Build Chain

A build chain is a set of software, generally with some sort of simple syntax or unique language, to assist in simplifying compiling and linking source code into machine code. Theoretically, one could forgo the use of a build chain and simply run compilers directly with the appropriate command-line flags, but this is extremely error-prone, excruciatingly slow, and generally not feasible. A build system, such as a hand-written Unix Makefile or Code::Blocks project file, will, based on the project's source code layout, contain the appropriate compiler invocations with the necessary command-line flags, and are made more easily accessible.

For instance, compilation with a Makefile would simply require typing "make" in a terminal in the appropriate directory, and compilation with a Code::Blocks project file would simply require clicking on the "Build" button in the Code::Blocks IDE.

0.4 Windows Installation Instructions

0.4.1 Simplified, Recommended Installation using Chocolatey

1. Visit https://chocolatey.org/ and follow its installation instructions under "Easy Install!".

2. After Chocolatey is installed, open a command prompt and run "choco install cmake mingw python" to download and install the sxc toolchain dependencies.

3. Re-open the command prompt, then run "python -m pip install cogapp".

4. Skip to and follow the instructions at 0.4.5.

0.4.2 CMake

Visit http://www.cmake.org/download/ and download a binary package for one's platform. When installing CMake, be sure to read the installer carefully and select "Add CMake to path". Otherwise, after installation, manually add '<CMAKE_INSTALL_DIRECTORY>\bin' to PATH 0.18.

0.4.3 Python

Visit https://www.python.org/downloads/release/python-342/ and download a binary package for one's platform. When installing Python, be sure to read the installer carefully and select "Add python.exe to path". Otherwise, after installation, manually add '<PYTHON_INSTALL_DIRECTORY> to PATH 0.18.

0.4.3.1 Cogapp

If one already has a Python2.7.x installation on one's system, one should check the PATH 0.18 environment variable to ensure that the Python3.4.x installation directory appears *before* the other Python install directory. This does not apply to most users, and can be usually safely skipped altogether.

After installing Python, simply run "python -m pip install cogapp" in a command prompt.

0.4.4 Unix/MinGW Makefiles

On Windows, the MinGW environment is required, as it provides the necessary build tools for Makefiles.

1. If installing MinGW for the very first time, download and run "mingw-get-setup.exe" from http://www.mingw.org/wiki/getting_started.

2. Go to the MinGW Installation Manager, and install the following packages:

   - mingw32-base
   - mingw32-automake

- mingw32-libstdc++
- mingw32-libintl
- msys-make

3. Add '<MINGW_INSTALL_DIRECTORY>\bin' to PATH 0.18.

0.4.5 SXC Cross-Compilation Suite

Double-click the included sxc.msi and follow the installer's steps. The selected installation directory's bin folder must be added to the PATH 0.18 environment variable.

Finally, if one installs the SXC Cross-Compilation Suite to a non-default directory (the default is assumed to be "C:
Program Files (x86)
SXC"), one must be mindful of the CMakeLists.txt provided in example projects and must modify the appropriate variables to point to the correct installation directory.

0.5 Mac OS X Installation Instructions

Coming soon.

0.6 Linux (Debian, Ubuntu) Installation Instructions

Run "dpkg -i <filename>" on the included sxc.deb as root, or with sudo privileges. All other dependencies will be downloaded and installed using the system's package manager (apt-get). Then run "python3 -m pip install cogapp".

0.7 Linux (Other) Installation Instructions

If the distribution can support ".deb" packages, use its recommended solution. Otherwise, use an archive manager to extract the contents of the package, and copy "usr" into "/". Required dependencies are: python3, python3-pip, cmake, make. Then run "python3 -m pip install cogapp".

0.8 OPTIONAL: Setting up a Build Chain

Eleven Engineering provides only the necessities for cross-compilation in order to make the toolset as flexible as possible. This means that the developer is free to use the IDE or text editor / workflow of their choosing to be as productive as possible. However, this also means that the chosen IDE or workflow must be itself flexible and must be able to be configured to use the SXC compiler and linker. This immediately excludes several solutions which are written to work only for a single platform.

An example of a cross-platform solution which meets these criteria is as follows:

- Unix/MinGW Makefiles, and almost any IDE / Text Editor

Examples of text editors and IDEs which may be used include:

- Sublime Text 0.8.1
- Code::Blocks 0.8.2
- Eclipse 0.8.3 CDT (note: requires a Java Runtime Environment)

Examples of solutions which do NOT meet these criteria are as follows:

- Visual Studio
- XCode

For beginners, the author recommends the use of Sublime Text as a text editor alongside Unix/MinGW Makefiles to build firmware. Code is written in the IDE, and a simple "make" command is used in a terminal / command prompt to build. The author personally uses the gVim text editor with a large assortment of plugins to provide IDE-like functionality, and uses Unix Makefiles to build.

One should note that there are a myriad of other build systems which will also work with CMake and the SXC cross-compilation suite, but Eleven Engineering will not provide support or setup instructions for them.

0.8.1 Sublime Text

Visit http://www.sublimetext.com/3 to download this text editor. Like gVim (and arguably Emacs), Sublime Text derives much of its functionality from optional plugins which extend its capabilities. Plugins can be easily installed using the package manager for Sublime Text, available at: https://sublime.wbond.net/installation.

0.8.2 Code::Blocks

Visit http://www.codeblocks.org/downloads/26 to download this IDE.

0.8.3 Eclipse CDT

Visit http://www.eclipse.org/cdt/downloads.php to download this IDE. Please note that Eclipse also requires a Java Runtime Environment to be installed on one's computer.

0.8.4 gVim

For those brave enough to enter the deep, dark depths of programming with gVim, the author's personal gVim configuration is available as a public Git repository at git@bitbucket.org:ew_eleveneng/vim_config.git. Read the included README for instructions on how to set up the more advanced plugins used.

0.9 Building a Project

The standard workflow is to run CMake once in the desired build directory (generally, the CMakeLists.txt for a simple project will not be modified over the project's lifetime), and then work with one's desired build chain. The development process generally consists of doing some programming, testing to see if the changes build and function properly, and then committing the changes to source control 0.17.

0.9.1 A Note on Included Examples

Various example projects are included with this guide to help the reader get started. The examples contain several headers, which are *NOT* part of the general library "sxc-lib", and which contain code common to many example projects. It is not recommended to use/copy these headers into one's own project's source directory until one understands how they work and differ from the main library.

0.9.2 Creating a Source Directory

There is little difference between setting up a CMake C project for a cross-compile to XInC2 and one for a native compile. In either case, one begins with a source directory containing their source code to compile, and a CMakeLists.txt file. For XInC2 cross-compilation, one only needs to be aware of the main.gen file, and the specific CMakeLists.txt text blocks that are required. A very

basic XInC2 project will closely resemble what is found at Examples/Sample_Project_Template, which should in fact be copied and modified accordingly in order to get started. One must rename the appropriate CMakeLists.txt for their board to CMakeLists.txt, and delete the other.

The correct selection of the CMakeLists.txt is *critical*; firmware that is configured and generated for one board type will *not* work on another type of board. Attempting to load firmware configured for one board to a different one will require a firmware reset (0.14) to "un-brick" the device.

The 'main.gen' file is a crucial part of the firmware image-building process. One should copy it verbatim into the root of one's project's source directory. This file should not be amended at all.

One should read through the copied CMakeLists.txt, make note of the comments, and make changes as appropriate, such as: changing the project name, or perhaps changing where header files are searched for. One could also write it from scratch, as long as one ensures that the CMakeLists.txt blocks which are marked "REQUIRED" are all included in the order provided.

Beyond this, one just needs to write C like one would for any other platform (with limitations: see Part II).

0.9.3 Running CMake

As long as the build directory is not deleted, the project's CMakeLists.txt does not change, and additional source files are not added, this step only needs to be done once for the project's lifetime. A massive benefit of CMake is that it greatly simplifies "out-of-source" builds, meaning that the source directory (which contains code) is never affected by compiling or building firmware; only the build directory will contain the results of compiling firmware. Please note that some source code must be present in the source directory in order for CMake to generate a project build.

1. Create a build directory that is separate from the project source directory (which contains the actual source code and auxiliary project files).
2. Go to this build directory.
3. If using MinGW on Windows, run 'cmake -G"MinGW Makefiles" <SOURCE_DIRECTORY>'. Otherwise, just run 'cmake <SOURCE_DIRECTORY>'.

The result is that the build directory becomes filled with CMake metadata files (which can be safely ignored) and, crucially, a Makefile that can be invoked to compile firmware.

0.9.4 Building Firmware with Unix/MinGW Makefiles

Once CMake has been successfully run in the build directory, building firmware with Unix/MinGW Makefiles is simple. To build firmware (ie: compile the code in the source directory), one opens the build directory in a terminal / command prompt and types "make" (or, if using MinGW on

Windows, "mingw32-make"). This calls all compilers, linkers, assemblers, and other programs in the correct order and with the correct flags in order to generate output files.

A successful build will result in (possibly several) static library files, some assembly files, and the binary firmware image, a file with extension ".hex", appearing in the build directory.

Part II

Writing Firmware for XInC2 in C

0.10 Key Restrictions and Differences from C Programming for Other Platforms

Like in the case of many embedded systems, because firmware compiled for XInC2 is flashed to the EEPROM and run "bare-metal", without any operating system in the way, a number of restrictions apply as to what C code is valid. Most C standard library functions that depend on a kernel or more elaborate CPU architecture are unavailable.

0.10.1 "Bytes" and All Integers are 16-bit

XInC2 is a 16-bit architecture. char, int, and int16_t all resolve to the same 16-bit signed integer type. uint16_t and size_t are equivalent to each other and are the unsigned counterpart of the former types.

Because of the limited range of 16-bit integers (0x0000 to 0xFFFF; unsigned: 0 to 65535; signed: -32768 to 32767), one may have to rethink how to store data. A theoretical example: a 32-bit unsigned integer can be easily utilized to store the price of an item in a store in cents, but the limited range of 16-bit integers may require splitting into multiple fields, like dollars and cents.

A more practical and common XInC2 example: in order to set a timer for a "human-scale" length of time (such as 1 second), multiple nested loops with separate indices which call sys_clock_wait() are necessary. This is because time is measured in terms of "system ticks" by the processor, which is typically run at ~ 74 MHz. To perform a single function call to wait for one second, one would have to pass an argument of ~ 74000000!

0.10.2 No Dynamic Memory Allocation

Without a kernel, the concept of dynamic/heap memory is meaningless. One cannot use C standard library functions such as malloc(), calloc(), or free() in one's code. All variables must be local, static, or extern. One should try to avoid using global variables as a work-around for this restriction whenever possible.

0.10.3 No Filesystem and No File Descriptors (ex. No stdout/stderr)

Without a kernel to define a file system or file descriptors, functions that inherently depend on writing to a file stream or to a file are unavailable. Notable examples include the entire printf() family, getchar() and related functions, and fopen() and friends.

To simulate printf(), one may connect the XPD hardware module to a personal computer and use XPD print functions to print to a terminal program, such as MTTTY.

0.10.4 Manual Hardware Binary Semaphore Management Required

XInC2's unique architecture implements multithreading in hardware: up to 8 threads may run simultaneously. In most cases, processor hardware resources are shared between threads, meaning that the behaviour of one thread may affect another in unexpected ways if it is not managed correctly. For instance, one thread may set a pin to an output while another thread is waiting to read an input on the same pin, leading to unexpected behaviour and a bug!

Unless separate threads access separate hardware resources altogether, the solution is to use the XInC2 binary semaphores in order to lock access to a hardware resource while a thread accesses it, preventing another thread from also affecting the hardware. Unless one can guarantee that a hardware resource will be configured correctly for a particular thread's uses (for example, it is never re-configured after program initialization), one should also code defensively and assume that the configuration is incorrect and must be redone every time the hardware resource is accessed in a thread. Finally, for the sake of performance (preventing threads from causing other threads to stall for too long), the semaphore should be acquired for as few instructions as is absolutely necessary.

A basic explanation and guide to binary semaphores and multithreaded programming is available in section 0.16.

0.11 sxc-lib

0.11.1 Overview

sxc-lib is the C library distributed with the SXC Cross-Compilation Suite which offers convenient access to hardware, such as GPIO and SPI, on a XInC2 board. It additionally adds some higher-level convenience functions, such as threading and semaphore utilization.

For the overwhelming majority of projects, one will only be utilizing functions and enumerations from GPIO.h, IOConfig.h, Semaphore.h, Thread.h, SFU.h, SPI.h, Timer.h, SystemClock.h, and possibly XPD.h. The API reference included alongside this guide lists and describes the functions provided by sxc-lib.

0.11.2 Hardware Configuration Flags

In most cases, hardware must be configured before using it; this is done by calling its corresponding set_config() function with the appropriate argument. The input config argument is always a set of flags which are bitwise-ORed together to form the desired configuration. The API reference included alongside this guide groups and lists these flags for each configurable hardware component.

While flag names are verbose, the meaning of every flag is not always immediately clear, and one should consult the XInC2 User's Guide to acquire a better understanding.

Part III

Loading Compiled Firmware onto a XInC2 Board

0.12 Prerequisite Software

- xdt (included in the SXC Cross-Compilation Suite)

0.13 Using xdt to Load Firmware

Successfully compiled firmware appears as a binary file ending with file extension ".hex" in the build directory. To load it to a XInC2 development board, run 'sxc-none-eabi-xdt -in <firmware_filename.hex> -port <COM_PORT_NUM>', where COM_PORT_NUM is the number of the COM port a plugged-in board is associated with on Windows. It can be found under Control Panel > Device Manager.

0.14 Fixes to Firmware Loading Issues

If the XInC2 dev board fails to ACK after receiving data, short pin PB0 to ground while loading firmware once. Subsequent attempts should not require this fix.

If a Ginger board fails to download firmware, short pins 1 and 3 on the XPD together, where pin 1 refers to the pin marked with a dot on the XPD PCB. After shorting these pins, begin the download and hit reset on the XPD. Subsequent attempts should not require this fix.

Part IV

Debugging Compiled Firmware

XInC2 will not support step-by-step program execution and debugging for the foreseeable future. Eleven Engineering recommends that the developer uses xpd_echo() functions to output variable values to an XPD terminal. Specifically, an XPD hardware module is connected between the development board and the developer's computer, a terminal program to read data from a USB port, such as MTTTY on Windows, is run, and function calls such as xpd_echo_int() can print values of variables for the developer to read and infer what is happening in the firmware.

A note on MTTTY: it is possible for the board to be placed in an infinite reset loop when connecting MTTTY to its COM port. To resolve this, disable RTS and DTS hardware flow control under the Flow Control menu.

Part V

General Programming Tools and Tips

This part will cover programming tools and advice which may prove to be highly helpful not only for programming for XInC2, but in general as well. None of the software mentioned here is strictly required for an actual XInC2 firmware build.

0.15 Good Coding Practices

- Never use global variables unless they are const, or they are shared across threads. If shared across threads, a variable should also have a corresponding semaphore to protect it from simultaneous access by multiple threads.
- Try to write code that is self-documenting. Unless one is writing a tutorial, code comments should explain *why* a particular section of code is doing something, not what it does.

0.16 Multithreading: Synchronization and Concurrency Using Binary Semaphores

0.16.1 Introduction and Justification

Programs are typically executed as a linear, serial sequence of instructions. This is known as a single thread of execution. Multithreading is the splitting of some tasks in a program into multiple, relatively independent threads of execution.

If the hardware on which the program is being executed contains redundant units, or additional CPU cores, multiple threads may execute instructions in parallel. This can result in a dramatic improvement in performance: if the threads are totally independent of each other, the speed of execution of the entire task becomes almost proportional to the number of threads, assuming that there is a separate execution unit or core for every thread.

Multithreading can also be utilized to improve the responsiveness of a program, not only its performance. A common use case is to operate the interface (such as a GUI, or polling a physical button, as in one of the example projects) on one thread, and run the primary program logic in another. In this way, the GUI remains responsive to inputs, even if the program is performing an intensive task.

0.16.2 Synchronization Problems

Multithreading does, however, pose a challenge when resources must be shared across threads. The crux of the issue is that multiple threads may simultaneously access and attempt to change data in shared memory, or may access and attempt to use shared hardware in different ways. The behaviour of the system in these situations is entirely unpredictable: which thread's actions are

executed first? What happens if the threads' actions are interleaved? What are the end results of the operations?

In the case of shared memory, whenever a variable is accessed from shared memory by a thread, it must be copied to a register before any arithmetic operations may be executed on it. If another thread modifies the variable in shared memory, any thread which has already copied the old variable value retains that value and thus acts on "stale" data, leading to incorrect program logic when it writes back the value to shared memory. Depending on which thread executes first, different results arise: this is known as a *race condition*.

Several mechanisms exist to prevent such issues from arising; however, fundamentally, all synchronization mechanisms work by preventing "critical sections" of code in different threads from executing simultaneously. Essentially, a mechanism is invoked in the thread's code, the shared variables or hardware are accessed or modified, and when the thread is done with them, it releases the mechanism. If a mechanism is currently engaged by a thread, any other thread which attempts to invoke the mechanism is blocked from execution until the thread which invoked it first releases the mechanism. Code which is written properly with such protection mechanisms is said to be "thread-safe".

In most cases, the operating system provides the protection mechanism and controls how and when threads are executed; in the case of XInC2, this is done directly in hardware without the use of an OS. The most notable synchronization mechanisms are semaphores and mutexes. Semaphores are themselves split into two categories: binary and counting. Because XInC2 only implements binary semaphores, only they will be discussed in this guide.

0.16.3 Multithreading and Binary Semaphores on XInC2

XInC2 implements multithreading directly in hardware, and up to 8 threads may execute simultaneously (technically, they are interleaved; more information is available in the XInC2 User's Guide). XInC2 additionally possesses 16 hardware binary semaphores, although one is reserved by sxc-lib to make thread running/stopping thread-safe; this leaves 15 (0 to 14) for the developer to use in their firmware.

A binary semaphore is an abstract mechanism that can either be "locked" or "unlocked"; semaphores begin existence "unlocked". When sem_lock() is called on a semaphore, it becomes "locked" and the calling thread continues execution as normal. If the semaphore is already "locked", the calling thread stops execution until the semaphore is "unlocked" by a call to sem_unlock() in another thread. Once the blocked thread begins executing again, it immediately locks the semaphore and continues execution. Calling sem_unlock() on an unlocked semaphore does nothing.

By surrounding a block of code involving shared variables or hardware with sem_lock() and sem_unlock(), one can guarantee that that block of code will never execute and thus will never access those resources if the semaphore is locked, and will always release the semaphore after it is done executing. One can then deduce that semaphores are a co-operative mechanism: *all* threads *must* lock and unlock semaphores themselves, before and after accessing shared resources, respectively. Any thread which does not call sem_lock() can access shared resources at any time, and

any thread which does not call sem_unlock() after calling sem_lock() can prevent all threads from accessing the resource in the future.

Semaphores should be acquired for the minimum duration necessary to access a shared variable and amend it as required. Locking a semaphore for longer than necessary can result in poor performance, because threads stall for excessive periods of time waiting on the semaphore to unlock, reducing the amount of parallel execution. Failing to unlock a semaphore will cause any thread which attempts to lock it to stall permanently!

Finally, sxc-lib offers certain "atomic functions", which internally lock and unlock a provided semaphore, and which do not need to be surrounded by calls to sem_lock() and sem_unlock(). In fact, calls to atomic functions *cannot* follow a lock of the very semaphore that they aim to use; this would result in total deadlock!

0.17 Source Control

Source control is the usage of a management program to track changes to a codebase over time and make it significantly easier to copy and modify over time. Use of source control will greatly assist in the development of any software project, but especially shines for those involving multiple collaborators, or a substantial amount of code.

Several such solutions exist: Git, Mercurial, CVS, and others. The author personally recommends Git, as it is arguably the most robust and scaleable, and is used industry-wide. Skill with Git, and source control in general, is a desirable and often lacking trait in programmers!

0.17.1 Common Terminology

Repository: the actual directory containing and corresponding metadata pertaining to a particular project being tracked by a source control program. A repository does not necessarily have to contain source code: any type of files may be tracked.

Commit: a snapshot of the current status (ie: file contents) of a repository.

0.17.2 Installing Git

Visit http://git-scm.com/downloads to download Git for one's platform of choice. Those who are more comfortable using a GUI to start off with Git may also wish to download a GUI client that uses Git as a backend. A notable one that is well-suited for beginners is SourceTree: http://www.sourcetreeapp.com/.

On a Linux distribution, Git should just be available through the system's package manager.

0.17.3 Using Git

A fantastic and substantial guide to Git can be found at http://git-scm.com/doc. The guide in this document will only cover the very basics to get one started with the Git command-line; GUIs will not be covered, and neither will particularly advanced topics, such as retroactively re-writing history in Git. One can also consult the manual included with Git for more details and options available to each Git command.

0.17.3.1 Git Concepts and Terminology: Working Directory, Index, and Commit Tree

To understand Git, one has to keep track of 3 distinct "areas" containing files and their changes: the working directory, the index, and the commit tree. One has to keep in mind that these are usually not synchronized as the programmer is working.

The working directory is simply the current state of the directory wherein the git repository is stored. Git will be able to compare differences between the working directory and either the index or a given commit, but otherwise does not "remember" the working directory.

The index or "staging area" consists of file changes that have been added via 'git add'. It can be thought of a sort of cache for the 'git commit' command. Like the working directory, Git will not "remember" the index, unless it is committed.

The commit tree is the remembered history of files and their changes over time that a Git repository is actually composed of, in terms of "commits". A commit can be thought of as a snapshot of the repository's index at a given time.

0.17.3.2 Git Concepts and Terminology: Commits and Branching

As mentioned before, commits are essentially snapshots of the index at given points in time. Commits have metadata associated with them, which constitutes the history of the Git repository. This metadata includes a timestamp and the commit's immediate ancestors (commits which immediately preceded the current one). Take note that a commit may have *multiple* ancestors.

This leads into another key concept, "branching", and into the realization that software development is not necessarily a strictly linear progression of commits (it is, after all, called a commit "tree", not a commit "line"!). Branching is the divergence of commits from a common ancestor: two different commits which share an ancestor form separate branches. This is extremely useful when multiple collaborators work on the project simultaneously; each developer may perform their work on their individual branches without affecting others'. Also, unlike real tree branches, Git branches may also merge back together, consolidating the changes done in each branch. Merges are what cause certain commits to have multiple ancestors.

Branches in Git can be thought of as implemented as pointers to commits, and for the most part, they point to the most recent commit along a distinct branch. The HEAD pointer is unique, and always refers to the current commit being pointed at.

0.17.3.3 Git Concepts and Terminology: Synchronization and Sharing

Git is known as a distributed version control system, meaning that every developer working on a project has their own full version of the repository. In the vast majority of cases, there is an agreed-upon central repository (generally stored on a server external to all of the developers') which each developer's changes are "pushed" to and "pulled" from in order to propagate code.

Pulling or fetching is the act of copying all commits and branches in the remote repository to the local repository, without overwriting any local changes done. The developer may then merge (with or without requiring an additional commit, depending on the progression of the commit tree) their local changes.

Pushing is the act of adding any additional commits a developer may have made in their local repository to the remote one. Pushing is a safe operation, in that any commits or mismatches in the local repository relative to the remote one will cause the push to fail, forcing the developer to pull the remote changes, merge their own changes properly, and then push.

0.17.3.4 Commonly-Used Commands: Fundamentals, Single Branch

- git init

  Initializes a repository in the current directory. Only run once and only when creating a new repository from scratch.

- git clone <path-to-repository>

  Copies the specified repository into a new directory with the same name as the repository. Only run once and only when creating a new repository from an existing one. Note that <path-to-repository> can be a path, or more commonly, a git URL to a remote repository.

- git add <filenames>

  Adds the changes (relative to the previous commit) made in the specified files to the index.

- git commit

  Saves the changes in the index as a commit, complete with user-supplied log message.

- git log

  Prints an ordered history of commits along the current branch.

- git checkout <commit>

  The state of tracked files in the working directory is made equivalent to the state of the specified commit.

0.17.3.5 Commonly-Used Commands: Multiple Branches

- git branch <branch-name>

  If the specified branch does not exist, then a new one is created, pointing at the currently checked-out commit.

- git checkout <branch-name>

  Tracked files in the working directory are made equivalent to the state of the commit that the specified branch points to, and Git switches branches to the specified one.

- git merge <branch-name>

  Merges the specified branch into the current one, consolidating the changes in both branches. If possible, no new commit will be created, but the current branch pointer will be "fast-forwarded" to the one being merged in. Otherwise, a merge commit is generated.

- git log –graph –oneline

  Prints a simple visualization of the repository's commit tree.

0.17.3.6 Commonly-Used Commands: Synchronization and Sharing

- git remote add <arbitrary-name> <path-to-repository>

  Adds the specified Git repository as a remote repository to synchronize with, which is referenced via the name provided. The path format is the same as for "git clone".

- git branch –set_upstream_to=<remote-name/branch-name>

  Causes the current branch to, when using "git pull" and "git push", to be synchronized with the specified branch in the specified remote repository.

- git pull

  Fetches all new commits on the remote server, and attempts to merge local changes on the current branch with those done on its "upstream" branch.

- git push

  Copies all new commits in the local repository to the remote server, and updates the current branch's upstream branch to point to the same commit as the current branch does. Will not work if the local repository is missing remote commits.

Part VI

Solving Installation Pitfalls

0.18 Updating the Windows PATH Environment Variable

If a call to any program used in the toolchain results in a message akin to "'<PROGRAM_NAME>' is not recognized as an external or internal command", then either the required program was not installed, or it was not added to the Windows PATH environment variable. To add it to PATH:

Control Panel > System > Advanced System Settings > Environment Variables > System Variables > Path

One should simply append the directory containing the program to this delimited list.