MuntsOS

Application Note #24: Development Environment Setup for x86-64 Windows 10/11

Revision 0
15 November 2025

by Philip Munts
***dba* Munts Technologies**
http://tech.munts.com

Introduction

This application note describes how to install the application software development environment for **MuntsOS Embedded Linux** (hereafter just **MuntsOS**) onto an x86-64 (*i.e.* AMD or Intel 64-bit CPU) computer running Windows 10 or 11.

While it is entirely feasible to run development tools targeting **MuntsOS** on a Windows computer in a Linux virtual machine, or in Windows Subsystem for Linux, these options impose a significant system administration burden, as you must install and maintain a full Linux distribution just to run a cross-toolchain.

Tarballs containing **MuntsOS** cross-toolchains running on 64-bit Windows 10 or 11 are available at:

https://repo.munts.com/muntsos/toolchain-win64

At time of writing, these cross-toolchains do not include support for GNU Modula-2, due to GCC Bug 92336.

Building a Cross-Toolchain for Windows

In the GCC cross-toolchain world, there are names for three different computers running potentially different operating systems:

Build is the computer you build your cross-toolchain on.
Host is the computer you run your cross-toolchain on.
Target is the computer you use your cross-toolchain to compile programs for.

Building a cross-toolchain that will run on Microsoft Windows was a terrible battle. Neither Crosstool-NG nor GNU Dev Tools for ARM are able at time of writing to successfully build a cross-toolchain (build=64-bit Windows, host=64-bit Windows, target=32- or 64-bit ARM Linux) using either Cygwin or MSYS2/Mingw-w64 (the two most popular environments for compiling and running Unix software on Windows). Crosstool-NG can ostensibly build a Canadian Cross (build=64-bit Linux, host=64-bit Windows, target=32- or 64-bit ARM Linux) toolchain in one go, but that also failed.

I eventually built a cross-compiler targeting Windows (build=64-bit Linux, host=64-bit Linux, target=64-bit Windows). Using that and the existing normal **MuntsOS** cross-toolchains (build=64-bit Linux, host=64-bit Linux, target=32- or 64-bit ARM Linux), I was finally able to manually build Canadian Cross **MuntsOS** cross-toolchains for Windows (build=64-bit Linux, host=64-bit Windows, target=32- or 64-bit ARM Linux).

Caveats & Limitations & Rants

Windows presents two main difficulties that make it nigh impossible to build a cross-toolchain targeting **MuntsOS** and difficult to even install one built elsewhere.

The first difficulty is the lack of symbolic links compatible with Unix operating systems such as Linux. Windows does provide symbolic links, but they are different enough that it becomes difficult to translate from Linux to Windows. One example of different functionality is that Linux is happy to create a symbolic link that points to a file that does not yet exist while Windows refuses to create a link to a non-existent target. Another complication is that Windows requires different kinds of links for different kinds of targets and some of them require administrative access to create.

The solution to this first difficulty is to replace all symbolic links in the cross-toolchain directory tree with the targets they point to. This is surprisingly easy to accomplish with:

```
rsync -avcqL
```

After replacing symbolic links with their targets, the size of the cross-toolchain directory tree will be somewhat larger, as there will be an extra copy of each file previously pointed to by a symbolic link.

The second and worst difficulty is that Windows and MacOS file systems are, by default, **case insensitive**, meaning upper case and lower case letters are not distinguished in file names. Only one of `aa`, `aA`, `Aa`, and `AA` can exist in a directory. If you create a file `aa` and subsequently open and write to `AA`, only the former will be present and will contain the data written to the latter.

Unix and Linux file systems are typically ***case sensitive***, meaning that `aa`, `aA`, `Aa`, and `AA` can all exist as distinct files in a directory. In a *rational* world this would not be a problem: *Rational* software developers would *never* deliberately create filenames differing only in letter case, both because such a policy prevents very obscure bugs (reading or writing the wrong file), and in recognition of the existence of more than one billion case insensitive Windows and MacOS computers that people actually use.

In the *real* world the situation is rather uglier. The Linux kernel source tree contains filenames differing only in letter case, making it impossible to unpack or checkout the Linux source tree to a typical Windows or MacOS file system. The GCC and/or binutils source trees also seem to contain filenames differing only in letter case, as may some of the library components included in the **MuntsOS** cross-toolchains.

The cross-toolchain distribution tarballs mentioned above in the first section of this document contain about 15 or so sets of filenames differing in case and present in the same directory, including the insane example of `linux/netfilter/xt_CONNMARK.h` that just includes its evil twin `linux/netfilter/xt_connmark.h`. This means that you cannot unpack one of the cross-toolchain distribution tarballs to a typical Windows file system without corrupting the cross-toolchain directory tree.

Windows from Windows 10 Build 1803 onwards provides a command **`fsutil.exe`** that can configure an *empty directory* for case sensitive filenames:

```
fsutil file SetCaseSensitiveInfo YourDestinationFolder enable
```

Files and directories subsequently created under **`YourDestinationFolder/`** will be case sensitive, and the Linux kernel could be checked out into **`YourDestinationFolder/`** or one of the cross-toolchains unpacked into **`YourDestinationFolder/`**.

*Note: The **`fsutil`** command above may not work unless WSL (Windows Subsystem for Linux) has been enabled. If you find that to be the case (pun untended) on your computer, you can issue the following command in a PowerShell window (opened in Administrative mode) to enable WSL:*

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

There does not appear to be any workable solution for any Windows older than Windows 10 Build 1803. It is *theoretically* possible to create a disk partition with MBR partition ID 7 and format said partition as a case sensitive Universal Disk Format file system using the following DOS command:

`format G: /fs:UDF` (Replace G: with your actual drive letter!)

Installation Procedure

Before Installing a Cross-Toolchain

In order to install and make use of one of the **MuntsOS** cross-toolchains for Windows 64 you will need to first install one of two Unix-like environments for Windows: Cygwin or MSYS2. Each of these environments provides a large collection of programs (such as **`ls`**, **`tar`**, or **`gmake`**) that originally came from Unix but have been compiled for Windows.

Both environments are collections of packages, each of which installs one or more programs. For Cygwin, the package manager is the Cygwin Installer. For MSYS2 the command line package manager is pacman.

*Tip: Alire (the Ada LIbrary Repository) automatically installs most or all of the MSYS2 packages that you will need to install and use the **MuntSOS** cross-toolchains for Windows. After you have installed Alire, you can use a batch file like alire.bat to set the **`PATH`** variable in a Windows command terminal.*

Installing a Cross-Toolchain

After Installing a Cross-Toolchain

You will also need to clone the following two **`git`** repositories to pick up some make include files necessary to compile programs with the **MuntsOS** cross-toolchains:

```
git clone https://github.com/pmunts/libsimpleio
git clone https://github.com/pmunts/muntsos
```

These should be cloned in your Windows home directory by default. If you wish to install them elsewhere, such as in C:/PROGRA~1/MuntsOS, you can set the **`LIBSIMPLEIO`** and **`MUNTSOS`** environent variables to the alternate checkout locations.