Building and Installing GNU `units` on Microsoft Windows with the MKS Toolkit

Edition 2 for `units` Version 2.16

Jeff Conrad

This manual is for building and installing GNU `units` (version 2.16) on Microsoft Windows with the PTC MKS Toolkit.

Table of Contents

Preface

This manual covers configuring, building, and installing GNU `units` from the MKS Korn shell on Microsoft Windows. The process runs much as it would on Unix-like systems, and much of what follows assumes that the installation will be in the same places as they would on Unix-like systems (e.g., `C:/usr/local/bin` for the executable). Most of the discussion implicitly assumes using Microsoft Visual Studio for compiling.

If Visual Studio is installed but Unix-like commands are not available, you can probably build `units` from the Windows command prompt using `Makefile.Win`—see *UnitsWin* for details.

A binary distribution for Windows is available, but if you use `more` or `less` as your pager, it is better to build `units` for MKS—see [Behavior of `PAGER`], page 5, for details.

The system on which the build was done had `/bin` as a symbolic link to `C:/Program Files (x86)/MKS Toolkit/mksnt`; with this approach, there is no need to change the first lines of any scripts in the units distribution.

The most recent build was for `units` version 2.16, using the MKS Toolkit for Developers version 10.0 and Microsoft Visual Studio 2015 on Microsoft Windows Professional 10 on 19 October 2017.

— Jeff Conrad (`jeff_conrad@msn.com`) 19 October 2017

Building and Installing `units`

Overview

On Unix-like systems, building and installing `units` is simple; just type

```
./configure; make; make install
```

On Windows—even if Unix-like utilities such as the MKS Toolkit are available—additional steps are usually needed. A more realistic procedure might be as follows:

1. Create a `config.site` file that specifies several parameters for `configure`. Alternatively, you can pass the parameters to `configure` at invocation.
2. Start an instance of the Korn shell with administrative privilege.
3. If you are using Microsoft Visual Studio, initialize the environment variables for Visual Studio with the `setvcvars` script:

   ```
   . ./setvcvars
   ```
4. Prepare the files needed to build `units` by running the configuration script:

   ```
   ./configure
   ```
5. Manually adjust `Makefile` if necessary.
6. Build the executable and support files:

   ```
   make
   ```
7. If the build is successful, install the package:

   ```
   make install
   ```

Some of the issues involved are discussed below.

Configuring configure

The configure script attempts to make the build process system independent. But on non–Unix-like systems, configure often needs some help. When using the MKS Toolkit on Windows, configure depends on the environment variables ac_executable_extensions and PATH_SEPARATOR. It is often easier to use the Microsoft Visual Studio C compiler cl directly rather than through the MKS wrapper cc; for this to happen, the variable CC must be set to cl or cl.exe.

The variables can be given to configure in several ways:

- The variables can be passed to configure at invocation as name–value pairs, i.e.,

  ```
  ./configure [name=value ...]
  ```

- The variables can be set and marked for export, e.g.,

  ```
  export ac_executable_extensions=".exe .sh .ksh"
  export PATH_SEPARATOR=";"
  ```

- The variables can be set in a site configuration script that is read by configure at invocation. Such a script might include

  ```
  ac_executable_extensions=".exe .sh .ksh"
  PATH_SEPARATOR=";"
  ```

  By default, the script is /usr/local/share/config.site. If you specify a location other than /usr/local/ for the installation with the --prefix option to configure, the configuration script is expected to be *prefix*/local/ share/config.site. If you wish to have a fixed location for the configuration script, you can do so with the CONFIG_SITE environment variable. For example, if you have a configuration script that you want read regardless of the --prefix option, you could give

  ```
  CONFIG_SITE="C:/usr/local/share/config.site"
  ```

  A more complete config.site might include

  ```
  ac_executable_extensions=".exe .sh .ksh"
  ac_ext=cpp
  prefix=C:/usr/local
  PATH_SEPARATOR=";"
  INSTALL="C:/usr/local/bin/install.exe -c"
  CC=cl.exe
  CFLAGS="-O2 -W3 -D_CRT_SECURE_NO_WARNINGS -nologo"
  CXX=cl.exe
  CXXFLAGS="-O2 -W3 -D_CRT_SECURE_NO_WARNINGS -nologo"
  ```

  (ac_ext, CXX, and CXXFLAGS are not needed for building units)

Customizing the Installation

By default, 'make install' installs units in subdirectories of /usr/local; you can specify a different location using the --prefix option. For example, if you want to install units in C:/Program Files (x86)/GNU, you might invoke configure with

```
./configure --prefix=C:/Progra~2/GNU
```

The Windows "8.3" short name is used because the installation process does not like spaces or parentheses in pathnames. The short name for C:/Program Files (x86) is usually as

shown, but can vary from system to system. You can find the actual short name on your system with the dosname command, e.g.,

```
dosname "C:/Program Files (x86)"
```

If you don't specify a prefix, or you specify a prefix without a drive letter, the installation will be on the same drive as the MKS Toolkit.

configure provides many other options for customizing the installation; typing

```
./configure --help
```

gives a summary of these options. Running configure is discussed in detail under the section "Running configure Scripts" in the GNU documentation for autoconf, available at http:// www.gnu.org/software/autoconf/.

Administrative Privilege

If you plan to install units in a location where you lack write permission, you'll need administrative permission for the installation and perhaps for the configuration and build (see ["install" Programs], page 4). The easiest way to do this is to start the shell by right-clicking on the shell icon (or a shortcut) from Explorer and using the Run as administrator option from the context menu.

Environment Variables for Visual Studio

Microsoft Visual Studio requires that several environment variables (e.g., PATH) be set to include numerous directories for a build from the command line. Visual Studio provides an option on the Windows Start Menu to run an instance of the Windows command interpreter with these variables initialized.

Initialization with the Korn Shell

The setvcvars script included in the units distribution will set these variables for the shell by running the batch file used for the Visual Studio command prompt, writing the variable values to the standard output, and reading them into the shell. For the values to persist, the script must of course be run in the current environment, e.g., '`source ./setvcvars`' or '`. ./setvcvars`'. These variables must be set for any command-line build with Visual Studio, so it may be helpful to copy the script to a directory that's in PATH (e.g., /usr/local/bin).

Adjustment for Different Visual Studio Installations

The location of the batch file and the values of the environment variables are installation specific; the setvcvars script assumes a standard installation of Visual Studio 2015 Express or Visual Studio 2015 Community. For a nonstandard installation or for a different version, the value of vsbatfile in the script may need to be modified. To find the appropriate value, go to the Windows Start Menu, find Visual Studio 20*xx* Developer Command Prompt for VS20*xx*, right click, and select Properties; the Target on the Shortcut tab should contain the proper path for the batch file.

On Windows 10, additional steps are needed to find the location of the batch file. Find Visual Studio 20*xx* on the Start Menu, click, right click on Developer Command Prompt for VS20*xx*, find More, right click, and select Open file location.

In the instance of File Explorer that opens, find the Developer Command Prompt shortcut, right click, and select Properties; the Target on the Shortcut tab should contain the proper path for the batch file.

"install" Programs

If you have an executable `install` program, you may get an error message to the effect of

```
cannot execute: The requested operation requires elevation
```

while running `configure` without elevated privileges on Windows Vista or later with User Account Control (UAC) enabled. If UAC is enabled, the system thinks executable programs whose names contain "install", "patch", "update", and similar always require elevated privilege, and will refuse to run them without this privilege.

If this happens, `configure` will simply use the `install-sh` script included with the `units` distribution. But if for some reason you wish to use your version of `install`, there are several ways to do so.

Running with Administrative Privilege

The easiest solution is to do the configure with a shell with administrative privilege, as discussed in [Administrative Privilege], page 3. After installation, testing should be done using a shell without elevated privilege.

Providing a Manifest File

An alternative is to tell UAC that elevated privilege is not required. To do this, create a manifest file containing

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <!-- Tell UAC that administrative privilege is not needed -->
        <requestedExecutionLevel level="asInvoker" uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

name it `install.exe.manifest`, and place it in the same directory as install.exe. Sometimes this has no effect; if this happens, adjust the modification times of the manifest and executable so they match.

The procedure is discussed at `https://github.com/bmatzelle/gow/issues/156`, and a similar discussion for GNU patch is given at `http://math.nist.gov/oommf/software-patchsets/patch_on_Windows7.html`.

Last access: 16 May 2016

Embedding a Manifest in the install Program

If you are using MS Visual Studio, an alternative to having the manifest file in the executable directory is to embed the manifest in the executable using the manifest tool `mt.exe`,

obviating the need to worry about the time stamps of the files. This is discussed in NIST link above; if the command is run from the shell, the semicolon must be escaped:

```
mt -manifest install.exe.manifest -outputresource:install.exe\;1
```

Microsoft describe manifests at https://msdn.microsoft.com/en-us/library/bb756929.aspx.

The Code Project also discusses UAC awareness: http://www.codeproject.com/Articles/17968/Making-Your-Application-UAC-Aware.

Fine Tuning Makefile

Behavior of PAGER

The MKS versions of more and less do not recognize +*n* as an option to display a file beginning at line *n*, so '`help` *unit*' from the units prompt will fail. If configure is able to detect the Toolkit by running mksinfo,

```
-DHAVE_MKS_TOOLKIT
```

is added to the DEFS in Makefile. If you have the MKS Toolkit and it somehow is not detected, you should add this manually.

MKS make and Suffix Rules

The MKS version of make ignores suffix rules in Makefile unless the line

```
.POSIX:
```

appears in Makefile before any suffix rules. This target is also required for the currency updater units_cur to run properly from Makefile. The configure script attempts to detect the Toolkit by running mksinfo, and if this succeeds, the .POSIX target is added. If you have the MKS Toolkit and it somehow is not detected, you should add this line manually.

Install Program

If the PATH at shell invocation uses the backslash as the path separator, and you have a BSD-compatible install program that is detected by configure, the backslashes may be removed, giving an incorrect Makefile entry something like

```
INSTALL = c:usrlocalbin/install.exe -c
```

Add the slashes to get

```
INSTALL = c:/usr/local/bin/install.exe -c
```

If you will always want to use the same installation program, you can specify it with the INSTALL variable—see [Configuring configure], page 2.

Giving a PATH with forward slashes in a file given by ENV will have no effect because configure unsets that variable, and the file will not be read.

Icons and File Association

Two icons are provided: `unitsfile.ico` and `unitsprog.ico`. The former is made the default icon for `units` data files, and the latter is embedded in the executable file by the build process. The latter also may be useful if you wish to create a shortcut to the `units` program. Both icons are copied to the same directory as the `units` data files.

The installation process associates `units` data files with the MKS graphical `vi` editor `viw`; double-clicking on the file icon opens the file for editing. The encoding is set to UTF-8.

MKS `units`

The MKS Toolkit includes a very old version of `units`; if the MKS executable directory is earlier in `PATH` than the installation directory for GNU `units`, a command-line invocation will run the MKS version. To ensure that you run GNU `units`, either change `PATH` so that GNU `units` is found first, or create an alias for GNU `units`.

Currency Definitions Updater

The script `units_cur` is used to update currency definitions; it requires Python (available from `https://www.python.org/`).

Installing Python

If you want to use the currency updater, install Python if it is not already installed; ensure that Python is installed *before* running `configure`. If you need to install Python, unless you have (or anticipate having) applications that depend on Python 2, the best choice is probably to install Python 3.

Python's location must be included in `PATH` so the shell can find it; the Python installer usually offers to do this.

When you first run `units_cur`, you may get a complaint about a missing module; for example,

```
ModuleNotFoundError: No module named 'requests'
```

If so, you will need to install the missing module. The easiest way to do this is with the `pip` command; for example,

```
pip install requests
```

If you have Python 2.7.9 or later or Python 3.4 or later, you should have `pip`, though you may need to upgrade to the latest version. If you do not have `pip`, you will need to install it manually; see the Python documentation or the Python website for instructions on how to do this.

Python and configure

The complete pathname in Makefile may contain backslashes; for example,

```
PYTHON = C:\Progra~1\Python\Python36/python.exe
```

The build will fail unless the backslashes are changed to forward slashes; for example,

```
PYTHON = C:/Progra~1/Python/Python36/python.exe
```

If a 32-bit version of Python is installed on a 64-bit Windows system, the Makefile entry may contain parentheses as well as backslashes, e.g.,

```
PYTHON = C:\Program Files (x86)\Python\Python36/python.exe
```

this will usually give a "syntax error" message when running configure. A Makefile entry such as this must be enclosed in single quotes for the build of units to succeed. The problem can be avoided by using the 8.3 equivalent of the Python installation directory in PATH, e.g.,

```
C:/Progra~2/Python/Python36/python.exe
```

An alternative is to install the 64-bit version of Python so that the installation directory will be C:\Program Files.

The backslashes can be avoided by passing PYTHON to configure at invocation, or by specifying it in config.site, e.g.,

```
PYTHON=C:/Progra~2/Python/Python36/python.exe
```

A disadvantage is that if the installation directory changes with a future version of Python, config.site will need to be manually updated. A better approach is to give the normal Unix/Linux pathname:

```
PYTHON=/usr/bin/python
```

This file need not exist; it simply tells the shell to use Python. Do not include the volume specifier (e.g., C:) or the .exe extension; if you do, the shell will assume that the path *does* exist, and will complain that it cannot find it.

Running the Currency Updater

Updating from the Command Line

If the location of units_cur is on your PATH, you can update the currency definitions by entering 'units_cur' from the command line; you will need elevated permission if you lack write permission on the file.

Reliable free sources of currency exchange rates have been annoyingly ephemeral, sometimes causing update attempts to fail. Accordingly, several different sources are now supported—see the units manual for details.

Automatic Updates

The easiest way to keep definitions updated is to create an entry in the Windows Task Scheduler. The Task Scheduler is fussy about the format for the action, which must be an executable file; an entry might look something like

```
C:\Windows\py.exe "C:\usr\local\bin\units\units_cur"
```

if the Python launcher is in C:\Windows and the script is in C:\usr\local\bin.