The Windows SDK Platform for WIT Software Projects

by Bob Wittrock

Revised, September 20, 2012

This document explains how to work with the port of WIT-related software projects to a platform that we are calling “Windows SDK”. This platform is actually a hybrid of two distinct sets of tools:

- Compilation, library creation, and linking of C++ and C programs are done using the Microsoft Windows SDK (Software Development Kit), a set of tools for developing application programs on Windows. The tools are invoked from the Windows command line and are available for free by downloading from the Microsoft website.
- The process of building programs (organizing their compilation, linking, etc.) is performed in an MSYS environment. MSYS (“Minimal SYStem”) is a collection of GNU utilities such as bash, make, etc. designed to enable programs to be built in a Unix-like style on Windows. It was created in order to be used with the MinGW compilers, but we're not using it that way here: we're using it with the Windows SDK compilers.

The Windows SDK platform is intended to be the means by which we will support WIT and its applications on Windows. It will replace the Cygwin and MinGW ports, which (as of this writing) still exist, but will be removed as soon as we can be sure no one is depending on them. It as been tested on Windows XP 32 bit and Windows 7 64 bit.

We chose this hybrid platform for the following reasons:

- MSYS allows us to extend and adapt our existing Makefiles to a Windows context. Our Makefiles are a complex system of inter-related files designed to be flexible and extensible. Creating and maintaining an alternative to them for Windows would have been burdensome and error prone.
- The Microsoft Windows SDK allows the Microsoft compilers, etc. to be invoked from a command line, which (in an MSYS context) allows them to be invoked from Makefiles.
- Our previous Windows ports, Cygwin and MinGW, are not supported for CPLEX. They were working on Windows 32 bit, but did not work on Windows 64 bit.

The WIT projects that I ported were: MCL, WIT, and WIT-J. The platform ID (used by the Makefiles) is “winsdk”.

To use this platform, you will need to carry out the following steps:

Step 1: Acquire the Tools

You'll need to acquire the following tools (in any order):

RTC

You'll need to have RTC with the WIT-Projects stream set up on your machine. We have other documentation on how to do that.

CPLEX

If you intend to build WIT with CPLEX embedded, you'll need to install the appropriate version of CPLEX on your machine. The `wit/winsdk/Makefile` specifically links in CPLEX 12.4, so that's the version you need. Either 32 bit or 64 bit is OK, depending on the architecture you are using. If you need help installing CPLEX, let me know.

Java

If you are going to build WIT-J, you'll need to install IBM Java. Instructions for this are included in the WIT-J documentation.

Microsoft Windows SDK

You'll need to install the Microsoft Windows SDK. Here are the steps that I took to do this:

- I went to the following web page:

  `http://www.microsoft.com/en-us/download/details.aspx?id=8279`

- This page says, “Download Center” and “Microsoft Windows SDK for Windows 7 and .NET Framework 4”. It shows a file name of “winsdk_web.exe”.
- I clicked DOWNLOAD.
- It asked where to put the download.
- I told it to put it on my desk top.
- It downloaded winsdk_web.exe to my desk top.
- I launched winsdk_web.exe.
- It brought up a window that said, “Windows SDK Set Up Wizard”.
- I clicked Next on several windows and agreed to the Licence Agreement and it installed.
- To verify the installation, I clicked
  Start >> Programs >> Microsoft Windows SDK v7.1 >> Windows SDK v7.1 Command Prompt
- This brought up the Windows SDK Build Environment Window, a yellow-on-black command line window, where I could enter familiar Windows commands (e.g. dir) and get the expected response. I could also enter the Windows SDK commands that our Makefiles will use: “cl”, “lib”, and “link”, but in this case, they just give error messages, due to a lack of arguments.
- We do not actually use the Windows SDK Build Environment Window in our Windows SDK build process, but it's good to be aware of it.
- I deleted winsdk_web.exe.

MSYS

You'll need to install MSYS. Here are the steps that I took to do this:

- I went to the following web page:

  http://www.mingw.org/wiki/Getting_Started

- This page says “Getting Started”. You can see that it pertains to MinGW, but actually the installer lets you select which portions of MinGW you want. In our case, we will select MSYS.
- Under “Graphical User Interface Installer”, I clicked “mingw-get-inst”.
- This took me to a Source Forge web page with a list of versions of mingw-get-inst. Above the list, it said:

  Looking for the latest version? Download mingw-get-inst-20120426.exe (662.7 kB)

- I clicked on that. Of course, there might be a newer version when you do it.
- It asked where to put the download.
- I told it to put it on my desk top.
- It downloaded mingw-get-inst-20120426.exe to my desk top.
- I launched mingw-get-inst-20120426.exe.
- It brought up a window that said, “Welcome to the MinGW-Get Set Up Wizard”.
- I clicked Next on several windows and agreed to the Licence Agreement.
- It eventually brought up a window that said “Select Components”.
- It had only “C Compiler” checked.
- I unchecked “C Compiler”.
- I checked “MSYS Basic System”.
- I clicked Next.
- With a few more windows, it installed.
- To verify the installation, I clicked Start >> Programs >> MinGW >> MinGW Shell.
- This brought up the MSYS Window, a green-on-black command line window, labeled “MINGW32”, where I could enter familiar Unix commands (e.g. ls -alF) and get the expected response.
- I deleted mingw-get-inst-20120426.exe.

makedepend.exe

If you want to use the “make depend” capability, you need get the makedepend.exe program and put it in a directory that's in your PATH on MSYS. It doesn't come with MSYS. I have a version of makedepend.exe that runs on Windows XP 32 bit and can send it to you.

Step 2: Get Acquainted with MSYS

The MSYS Window is the environment in which our Makefiles for the Windows SDK platform are designed to operate. Here are some important points about the MSYS environment:

- As you can see by issuing ls commands, the MSYS environment, places you in what looks like a Unix-style file system:
  - Pathnames are delimited with forward slashes.
- On the other hand, like Windows, names are case insensitive.
- In fact, the MSYS file system is actually a directory in the Windows file system.
  - Specifically, the root directory of MSYS, "`/`", is following directory on Windows:
    `C:\MinGW\msys\1.0\`
- Furthermore, the Windows file system shows up as a directory in the MSYS file system.
  - Specifically, the "`C:\`" directory on Windows is the following directory in MSYS:
    `/C/`
- In this way, all MSYS files can be accessed from Windows and vice versa.
- There's one caveat: MSYS (like Unix) is seriously intolerant of blank spaces in file names and directory names, and Windows uses blanks in some of its directory names.
  - To work around this problem, use the "dir /x" command from the Windows command prompt. This lists the files as usual, but if any file or directory name has blanks or is longer than 8 characters, it also shows an alternate name with 8 characters and no blanks. So when I issue "dir /x" from C:\, the directory "Program Files" shows up along with its alternate name "PROGRA~1". I can then safely use "PROGRA~1" in place of "Program Files" in the MSYS environment. We will need this.
- MSYS creates a home directory for your MSYS user name, which is your Windows user name. The home directory is located under the /home directory; my MSYS home directory is: /home/wittrock.

Step 3: Set Up Your MSYS Environment

There is some set up that needs to be done to your MSYS environment before our Makefiles will work correctly in it:

RTC Local Workspace

You'll need to load an RTC local workspace for WIT-Projects somewhere on your machine where you will access it from MSYS. The RTC operations would be done from Windows, while Makefile operations are going to be done from MSYS. (It's conceivable that you could use the RTC Command Line Interface from MSYS, but I haven't tried it.) There are two alternative approaches to this:

(a) You can place the local workspace in the MSYS file system. This allows you to refer to it in the "natural" way when doing Makefile operations, but you need to refer to the `C:\MinGW\msys\1.0\` directory when doing RTC operations.
(b) You can place the local workspace in the Windows file system. This allows you to refer to it in the "natural" way when doing RTC operations, but you need to refer to the `/C/` directory when doing Makefile operations.

I use approach (a): For RTC purposes, my local workspace is the following directory:

```
C:\MinGW\msys\1.0\home\wittrock\wit-projects
```

For Makefile purposes, it's the following directory:

```
/home/wittrock/wit-projects
```

MSYS Environment Variables

You'll need to set a number of MSYS environment variables. I suggest setting them in your MSYS login profile, in my case:

```
/home/wittrock/.profile
```

Here are the environment variable settings that I needed to do on Windows XP, 32 bit:

```
export PATH="/C/Program Files/Microsoft Visual Studio 10.0/Common7/IDE:$PATH"

export PATH="/C/Program Files/Microsoft Visual Studio 10.0/VC/Bin:$PATH"

export INCLUDE="C:\Program Files\Microsoft Visual Studio 10.0\VC\include;$INCLUDE"

export LIB="C:\Program Files\Microsoft SDKs\Windows\v7.1\Lib;$LIB"

export LIB="C:\Program Files\Microsoft Visual Studio 10.0\VC\lib;$LIB"

export WIT_CPLEX_HOME=/C/Progra~1/IBM/ILOG/CPLEX_Studio124/cplex

export WIT_HOME=~/wit-projects

export JAVA_HOME=/C/Progra~1/IBM/Java60

export PLATFORM=winsdk
```

The first 5 exports above are needed to make the Windows SDK work.

On Dan's Windows 7, 64 bit machine, the first 5 became:

```
export PATH="/C/Program Files/Microsoft Visual Studio 10.0/Common7/IDE:$PATH"

export PATH="/C/Program Files/Microsoft Visual Studio 10.0/VC/Bin/amd64:$PATH"

export INCLUDE="C:\Program Files\Microsoft Visual Studio 10.0\VC\include;$INCLUDE"

export LIB="C:\Program Files\Microsoft SDKs\Windows\v7.1\Lib\x64;$LIB"

export LIB="C:\Program Files\Microsoft Visual Studio 10.0\VC\lib\amd64;$LIB"
```

On Donna's Windows 7, 64 bit machine, the first 5 became:

```
export PATH="/C/Program Files (x86)/Microsoft Visual Studio 10.0/Common7/IDE:$PATH$"

export PATH="/C/Program Files (x86)/Microsoft Visual Studio 10.0/VC/Bin/amd64:$PATH$"

export INCLUDE="C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\INCLUDE;$INCLUDE"

export LIB="C:\Program Files\Microsoft SDKs\Windows\v7.1\Lib\X64;$LIB"

export LIB="C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\Lib\amd64;$LIB"
```

On different Windows configurations, you may need to use different values. Note the following:

- The PATH entries are interpreted by MSYS. The pathnames are given in MSYS style, except that blanks are tolerated here for some reason.
- The INCLUDE and LIB entries are interpreted by the Windows SDK tools. The pathnames are given in Windows style.
- To see what values Windows SDK uses for PATH, INCLUDE and LIB, bring up the Windows SDK Build Environment Window. Click:
  Start >> Programs >> Microsoft Windows SDK v7.1 >> Windows SDK v7.1 Command Prompt
  - Then issue: `echo %PATH%`, etc.
  - Unfortunately, this approach will show you more than just the values you need for our purpose (especially in PATH). I suggest selecting 5 values that resemble the 5 shown in this document.

The WIT_CPLEX_HOME variable is only needed if you are going to build WIT with CPLEX embedded. If you are doing so, WIT_CPLEX_HOME needs to be set so that `$WIT_CPLEX_HOME/include/ilcplex` contains cplex.h and `$WIT_CPLEX_HOME/lib` contains the CPLEX library somewhere in its subdirectory structure. This variable is interpreted by the Makefiles and is given as an MSYS pathname, no blanks allowed.

If you are on Windows 64 bit and are building WIT with CPLEX embedded, one additional environment variable needs to be set:

```
export WIT_CPLEX_LIB_SUBDIR=x64_windows_vs2010/stat_mta
```

This is the subdirectory in which the CPLEX library can be found. It only includes the part of the directory path after /lib/. This is an override to a default value that's specified in the config directory. The default for our Windows SDK platform is for Windows 32 bit, so this override is needed for the 64 bit case. This variable is interpreted by the Makefiles and is given as an MSYS pathname, no blanks allowed.

The last 3 variables WIT_HOME, JAVA_HOME, and PLATFORM are only used by WIT-J and otherwise not needed. They are explained in the WIT-J documentation file: Set-Up-WIT-J.txt. These variables are interpreted by scripts and are given as MSYS pathnames, no blanks allowed.

Step 4: Build and Test the WIT Software Projects

Building and testing MCL, WIT, and WIT-J on our Windows SDK platform is all done in the MSYS environment. The instructions for doing this will assume that your RTC local workspace for WIT Projects is located in the MSYS file system as:

```
~/wit-projects
```

MCL

To build MCL on the Windows SDK platform, do the following:

- `cd ~/wit-projects/mcl/winsdk`
- `make exe_type=released all`

Notes:

- You can use any exe_type.
- The “all” target builds the MCL Unit Tester, which also causes the MCL library to be built.
- You can use the unitTest.exe target instead.
- If you use unitTest as the target, it stops with an error message that I wrote.
- You can use the -j[N] argument (e.g., -j4) to get N asynchronous compiles. This is a faster build, if your machine has more than one core.
- “make clean” also works on this platform.
- “make depend” works on this platform, if you have put a working version of makedepend.exe in your PATH.

To test MCL on the Windows SDK platform, do the following:

- `cd ~/wit-projects/mcl/winsdk`
- `unitTest`

This produces lots of output, concluding with:

```
All tests completed successfully
```

WIT

To build WIT on the Windows SDK platform, do the following:

- `cd ~/wit-projects/wit/winsdk`
- `make exe_type=released all`

Notes:

- You can use any exe_type.
- The “all” target builds the WIT Stand-Alone Executable, which also causes the WIT library to be built.
- You can use the wit.exe target instead.
- You can use the -j[N] argument.

To test the WIT Stand-Alone Executable on the Windows SDK platform, do the following:

- `cd ~/wit-projects/wit/winsdk`
- `wit {filename}`

where `{filename}` is the name of a WIT parameters file, as usual.

To test WIT in API mode, do the following:

- `cd ~/wit-projects/samples`
- `cl     apiAll.c -Za -EHs -nologo -I../wit/src ../wit/winsdk/wit.lib`
- `apiAll`
- `cl -Tp msgAll.C -Za -EHs -nologo -I../wit/src ../wit/winsdk/wit.lib`
- `msgAll`

Notes:

- The run of apiAll produces lots of output, concluding with:

  ```
  Normal termination of apiAll.
  ```

- The run of msgAll produces lots of output, concluding with:

  ```
  WIT0098I WIT function witGetDevMode entered.
  ```

WIT-J

To build and test WIT-J, follow the instructions in:

```
~/wit-projects/witj/doc/Set-Up-WIT-J.txt
```

Step 5: Build and Run Application Programs

Building a WIT Application Program:

To build a C++ WIT application program called foo.C in your MSYS environment, do the following:

```
cl -Tp foo.C -Za -EHs -nologo -I ~wit-projects/wit/src ~wit-projects/wit/winsdk/wit.lib
```

- The -Tp argument tells cl that the next argument is the name of a C++ file, even if it doesn't have a .cpp suffix.
- The -Za argument tells cl to treat your source code as ANSI C++ and not Microsoft C++.
- The -EHs argument tells cl to use an exception handling convention that seems to match the way WIT works.
- The -nologo argument tells cl not to display a copyright banner.

You can also build WIT application programs in Windows without the MSYS environment. There are several alternative ways to do this:

- Build the application from the Windows SDK Build Environment Window:
  - Click:
    Start >> Programs >> Microsoft Windows SDK v7.1 >> Windows SDK v7.1 Command Prompt
  - This brings up the Windows SDK Build Environment Window.
  - Then issue the “cl” command given above, but identify the WIT directories using Windows path names.
- Or, build the application from an ordinary Windows command line window:
  - First, you will need to do the first 5 environment variable settings described under “MSYS Environment Variables”, but do them in the Windows environment, using Windows path names.
  - Then issue the “cl” command given above, but identify the WIT directories using Windows path names.
- Or, use the Microsoft Visual Studio IDE, if you have it:
  - Use the “cl” command given above as a guideline for telling the IDE where to look for wit.h and wit.lib.
  - We haven't tried this, but I think it should work.

Running a WIT Application Program

Once you've built your WIT application program, you do not need MSYS or the Windows SDK or our environment set up to run it. The application should be able to run on Windows by itself, subject to one restriction: If your application program uses WIT with CPLEX embedded, your PATH variable must include the directory that contains cplex124.dll. Fortunately, the process of installing CPLEX puts this in your PATH automatically. But if you deploy your application elsewhere, the cplex124.dll file needs to be present and its directory must be in the PATH. Of course, you need to consider the CPLEX licensing issues in this case, just as you would on Linux, where the CPLEX library is buried inside the WIT library.

Building and Running a WIT-J Application Program:

See:

```
~/wit-projects/witj/doc/Demo1-Deployment.txt
```

So that's basically what you need to know to use the existing ports of WIT software projects to the Windows SDK platform.

Porting Other WIT Software Projects to Windows SDK

As of this writing, the following WIT projects have been ported to the Windows SDK platform:

- MCL
- WIT
- WIT-J

The remaining WIT projects on RTC have not been ported to the Windows SDK platform:

- ESO2
- SCE
- Scenario
- WIT-M
- WitUtil
- WitViz

And in the future, we may create other WIT application programs using this Makefile system that need to be ported to the Windows SDK platform.

Here are some guidelines for how to port the Makefiles for a WIT software project to the Windows SDK platform:

If the program is a WIT-J application program, then it's in Java and all you need to do is to have your program access the witj.jar and witj.dll files on Windows. No new Makefiles are required. This applies to WitViz and WIT-M.

Otherwise, we can assume that the program is a C++ WIT application program. To give you an idea of how to port a C++ project to the Windows SDK platform, I will explain what I did for MCL:

How MCL Was Ported to Windows SDK

- I created and worked out `config/p_winsdk.mk`.
  - You don't have to do this; it's done once and for all.
- I created `mcl/src/appl_unix.mk`.
  - The purpose of this file is to be the "Make file for aspects of building MCL that follow a Unix-like convention, but are otherwise platform-independent".
  - Initially, it contained only the above statement, as a comment.
- To each existing platform Makefile for MCL (`mcl/linux/Makefile`, `mcl/zlinux64/Makefile`, etc.), I added:

  ```
  include ../src/appl_unix.mk
  ```

  just after:

  ```
  include ../src/appl.mk
  ```
- Then I took all aspects of appl.mk that were inappropriate for Windows SDK and moved them to appl_unix.mk. These were:
  - The unitTest target.
  - The libmcl.a target.
- Then I created the build directory: `mcl/winsdk`.
- Then I created the platform-specific Makefile: `mcl/winsdk/Makefile`.
- Initially I did this as a copy of the familiar Linux Makefile: `mcl/linux/Makefile`.
- Then I made the appropriate modifications for Windows SDK:
  - Removed line: `include ../src/appl_unix.mk`
  - Removed the dependency list generated by make depend on Linux.
  - Altered line: `platform = winsdk`
  - Modified `ds_comp_std_flags`, as needed.
- Then I added targets corresponding to those in appl_unix.mk, but done appropriately for Windows SDK:
  - The MCL library, mcl.lib, is built using the Windows SDK "lib" command.
    - For documentation on the "lib" command, see: http://msdn.microsoft.com/en-us/library/e17b885t%28v=vs.100%29.aspx
  - The MCL unit tester, unitTest.exe, is built using the Windows SDK "link" command.
    - For documentation on the "link" command, see: http://msdn.microsoft.com/en-us/library/t2fck18t%28v=vs.100%29
  - I also added a "unitTest" target to generate an error message. (Not strictly necessary.)
- Then I issued "`make depend`" from the `mcl/winsdk` directory to generate the dependencies for MCL on the Windows SDK platform.
- Then I built and tested MCL, as described in this document.
  - This, of course, is when any actual C++ source code porting needs to be done. As I recall, I did need to make some code changes in MCL, WIT, and WIT-J. Nothing too bad.
- That was it for MCL.

The porting of WIT and of WIT-J were similar: In each case, I took the parts of appl.mk that were inappropriate for Windows SDK and put them into appl_unix.mk. Then I created the platform Makefile. I think this approach should always be used, if possible.