Using the Visual C++ IDE to edit/build AVR projects via WinAVR

I have developed applications targeted for AVR microcontrollers and the Windows OS. I use Microsoft's Visual C++ for Windows applications. The Visual C++ IDE has some features—integrated source version control, project source browser, IntelliSense, column editing—that I especially like, so I was interested in trying it out to edit and build AVR projects via WinAVR.

This is the procedure I followed (I originally referred to http://hubbard.engr.scu.edu/embedded/avr/msvc_make/msvc_make.html):

1. Created a makefile.
2. Modified the makefile build recipes to translate the format of avr-gcc diagnostic messages to be compatible with the Visual C++ IDE.
3. Created a Visual C++ Makefile project (the Makefile Project Wizard was handy).
4. Configured the Visual C++ project build command line and IntelliSense properties.
5. Manually added source files to the Visual C++ project.
6. Simulated/debugged the project application, and programmed/ran/debugged on the target.

I have summarized my findings and solution here.

Makefile

After creating a processor- and speed-specific makefile (I used *Mfile - A Makefile generator for AVR-GCC* (http://www.sax.de/~joerg/mfile/)), I modified the build recipes to translate the format of *avr-gcc* diagnostic messages to be compatible with Visual C++, so that I could take advantage of the IDE feature that sends my cursor to the offending line of source code when I double-click a build warning or error message.

Diagnostic message format translation is described in detail in the Appendix.

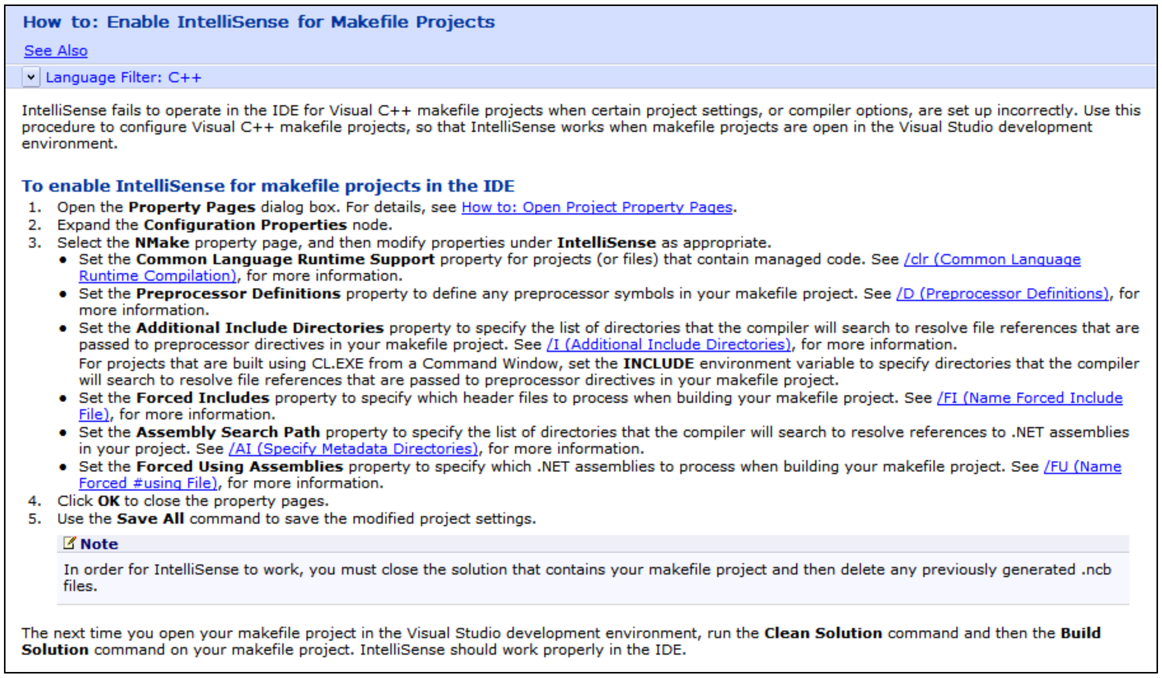
Here are the makefile modifications I made:

```
. . .

CC_DIAGS_XLATE = 2>&1 | sed -e "s_:\([0-9]*\):\([0-9]*\):_\(\1,\2\):_"  \
                            -e "s_:\([0-9]*\):_\(\1\):_"

. . .

# Link: create ELF output file from object files.
.SECONDARY : $(TARGET).elf
.PRECIOUS : $(OBJ)
%.elf: $(OBJ)
       @echo
       @echo $(MSG_LINKING) $@
       $(CC) $(ALL_CFLAGS) $^ --output $@ $(LDFLAGS) $(CC_DIAGS_XLATE)


# Compile: create object files from C source files.
$(OBJDIR)/%.o : %.c
       @echo
       @echo $(MSG_COMPILING) $<
       $(CC) -c $(ALL_CFLAGS) $< -o $@  $(CC_DIAGS_XLATE)


# Compile: create object files from C++ source files.
$(OBJDIR)/%.o : %.cpp
       @echo
       @echo $(MSG_COMPILING_CPP) $<
       $(CC) -c $(ALL_CPPFLAGS) $< -o $@  $(CC_DIAGS_XLATE)


# Compile: create assembler files from C source files.
%.s : %.c
       $(CC) -S $(ALL_CFLAGS) $< -o $@ $(CC_DIAGS_XLATE)


# Compile: create assembler files from C++ source files.
%.s : %.cpp
       $(CC) -S $(ALL_CPPFLAGS) $< -o $@ $(CC_DIAGS_XLATE)


# Assemble: create object files from assembler source files.
$(OBJDIR)/%.o : %.S
       @echo
       @echo $(MSG_ASSEMBLING) $<
       $(CC) -c $(ALL_ASFLAGS) $< -o $@ $(CC_DIAGS_XLATE)

. . .
```

Visual C++ Project Settings

In addition to specifying the build (*Build*, *Rebuild All*, and *Clean*) command lines, I modified a couple of the configuration properties of my Visual C++ project to take advantage of the IntelliSense feature.

First, I followed the instructions in this relevant How-to from the Visual C++ documentation.



How to: Enable IntelliSense for Makefile Projects

See Also

Language Filter: C++

IntelliSense fails to operate in the IDE for Visual C++ makefile projects when certain project settings, or compiler options, are set up incorrectly. Use this procedure to configure Visual C++ makefile projects, so that IntelliSense works when makefile projects are open in the Visual Studio development environment.

To enable IntelliSense for makefile projects in the IDE

1. Open the **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Select the **NMake** property page, and then modify properties under **IntelliSense** as appropriate.
   - Set the **Common Language Runtime Support** property for projects (or files) that contain managed code. See /clr (Common Language Runtime Compilation), for more information.
   - Set the **Preprocessor Definitions** property to define any preprocessor symbols in your makefile project. See /D (Preprocessor Definitions), for more information.
   - Set the **Additional Include Directories** property to specify the list of directories that the compiler will search to resolve file references that are passed to preprocessor directives in your makefile project. See /I (Additional Include Directories), for more information.
     For projects that are built using CL.EXE from a Command Window, set the **INCLUDE** environment variable to specify directories that the compiler will search to resolve file references that are passed to preprocessor directives in your makefile project.
   - Set the **Forced Includes** property to specify which header files to process when building your makefile project. See /FI (Name Forced Include File), for more information.
   - Set the **Assembly Search Path** property to specify the list of directories that the compiler will search to resolve references to .NET assemblies in your project. See /AI (Specify Metadata Directories), for more information.
   - Set the **Forced Using Assemblies** property to specify which .NET assemblies to process when building your makefile project. See /FU (Name Forced #using File), for more information.
4. Click **OK** to close the property pages.
5. Use the **Save All** command to save the modified project settings.

Note

In order for IntelliSense to work, you must close the solution that contains your makefile project and then delete any previously generated .ncb files.

The next time you open your makefile project in the Visual Studio development environment, run the **Clean Solution** command and then the **Build Solution** command on your makefile project. IntelliSense should work properly in the IDE.

[Source: http://msdn.microsoft.com/en-us/library/ms173379%28v=vs.80%29.aspx]

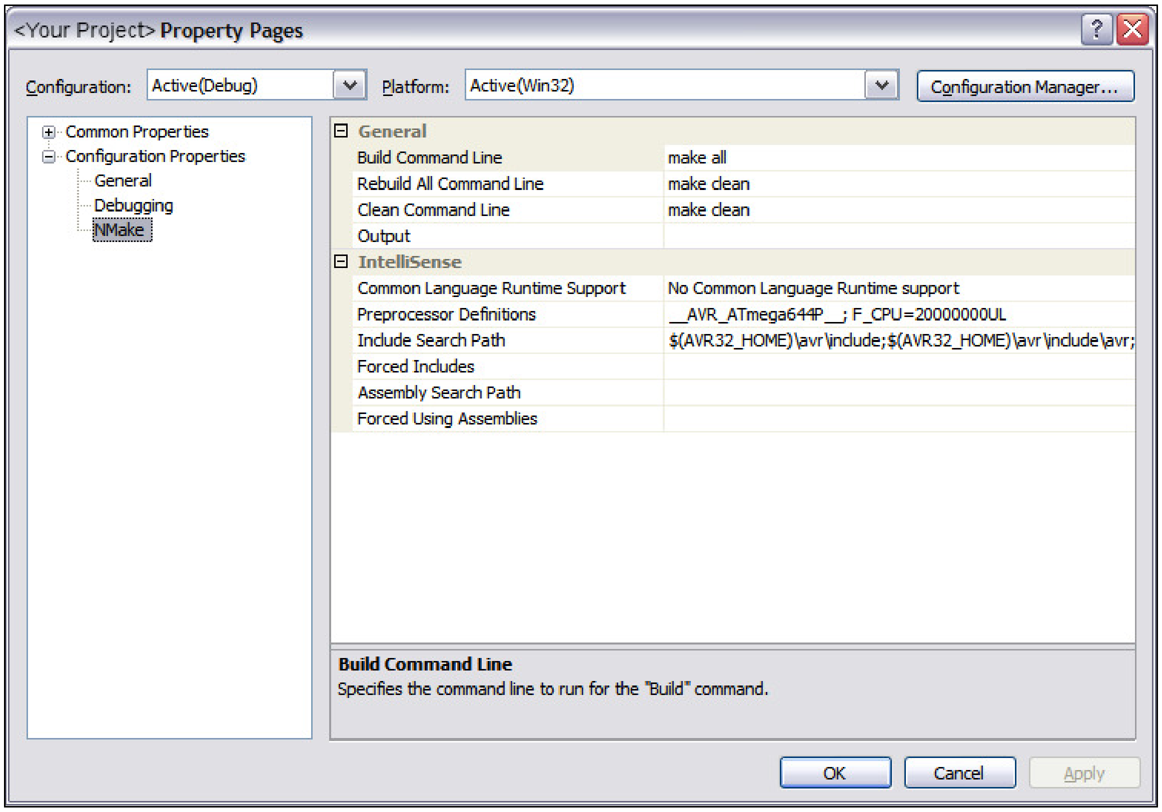
IntelliSense needs to know where to find relevant source files, so I set the *Additional Include Directories* to include all of the WinAVR AVR include directories:

```
$(AVR32_HOME)\avr\include;$(AVR32_HOME)\avr\include\avr;$(AVR32_HOME)\a
vr\include\compat;$(AVR32_HOME)\avr\include\util
```

(The `AVR32_HOME` environment variable was created by my WinAVR installation, so I used it instead of explicitly typing the full path.)

In order to enable IntelliSense to do its best to format the text in the text editor window, I set the *Preprocessor Definitions* to include all of the definitions that the avr-gcc compiler would see at build time, but were not already explicitly defined in the source files (e.g. `-mmcu` and `-D` command-line options).

Here is what I ended up with.



General	
Build Command Line	make all
Rebuild All Command Line	make clean
Clean Command Line	make clean
Output	
IntelliSense	
Common Language Runtime Support	No Common Language Runtime support
Preprocessor Definitions	__AVR_ATmega644P__; F_CPU=20000000UL
Include Search Path	$(AVR32_HOME)\avr\include;$(AVR32_HOME)\avr\include\avr;
Forced Includes	
Assembly Search Path	
Forced Using Assemblies	

Build Command Line
Specifies the command line to run for the "Build" command.

Simulate/Run/Debug Application

After building the project application binary with WinAVR via Visual C++ IDE, I used AVR Studio (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725) to simulate the code, and to program/run it on the target device, discovering/fixing bugs along the way.

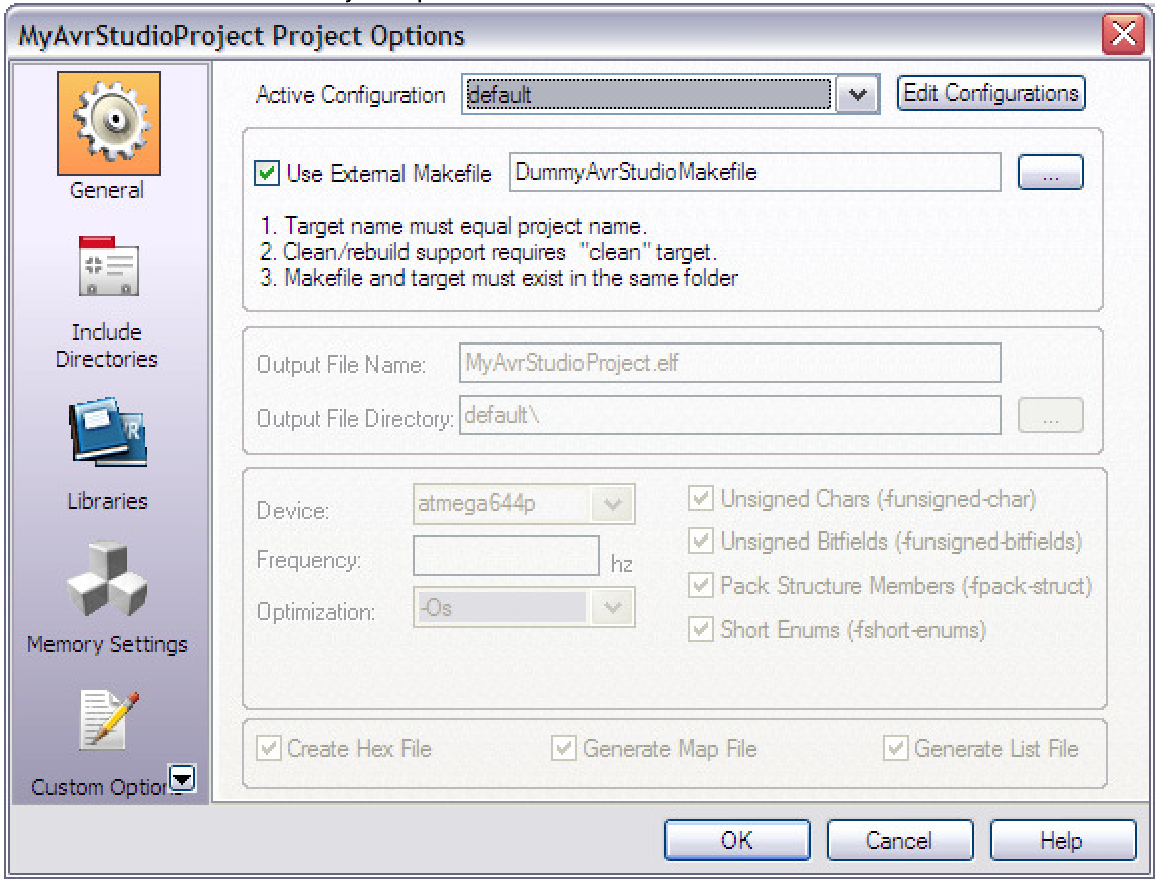
Note: Alternative processes exist (e.g. http://www.avrfreaks.net/wiki/index.php/Documentation:AVR_GCC/AVR_GCC_Tool_Collection and http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_GCC_Toolchain) for this and all AVR software development phases.

In summary, I performed the following procedure to debug with AVR Studio:

1. Created an AVR Studio External Makefile project.
2. Created a makefile to perform a dummy build.
3. Added sources to the project.
4. Performed a dummy build via the Build/Build menu command.
5. Ran the program via the Debug/Start Debugging menu command.
6. Discovered a bug.
7. Stopped execution with Debug/Reset (Not Debug/Stop Debugging).
8. Changed source with Visual C++ IDE editor.
9. Rebuilt executable via Visual C++ IDE.
10. Reloaded executable with AVR Studio.
11. Repeated Steps 5 through 10 as required.

To debug with AVR Studio I needed the Debug/Start Debugging menu command to be enabled. However, as a prerequisite for enabling the Debug/Start Debugging menu command, AVR Studio apparently (obviously) needed to successfully build the project. Apparently AVR Studio needs a project to “do its thing”, so I created an External Makefile project and associated makefile to perform a dummy build to expose it to the already-existing executable file. I also added my source files to the project for my navigation convenience, and to insert breakpoints prior to execution.

Here are the AVR Studio Project Options and makefile I used:


MyAvrStudioProject Project Options

Active Configuration: default

Edit Configurations

☑ Use External Makefile: DummyAvrStudioMakefile

1. Target name must equal project name.
2. Clean/rebuild support requires "clean" target.
3. Makefile and target must exist in the same folder

Output File Name: MyAvrStudioProject.elf

Output File Directory: default\

Device: atmega644p

Frequency: hz

Optimization: -Os

☑ Unsigned Chars (-funsigned-char)

☑ Unsigned Bitfields (-funsigned-bitfields)

☑ Pack Structure Members (-fpack-struct)

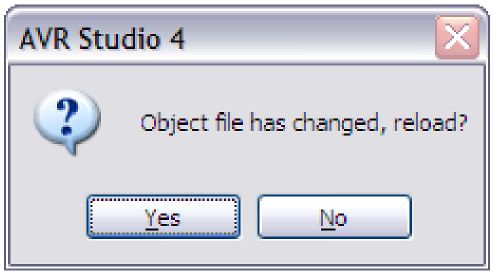
☑ Short Enums (-fshort-enums)

☑ Create Hex File ☑ Generate Map File ☑ Generate List File

DummyAvrStudioMakefile:

```
# Dummy makefile for AVR Studio to force it to enable the Debug menu command
# by believing that it has made a successful target.

all: ;
```

I streamlined the typically cyclical edit/build/debug process a little by relying on AVR Studio to recognize when a new executable file had been externally built while it remained in Debug mode and connected to the target. Specifically, rather than the Debug/Stop Debugging command, I used Debug/Reset before editing/rebuilding in the Visual C++ IDE, then waited for AVR Studio to display the following dialog after detecting the newly built executable.


AVR Studio 4

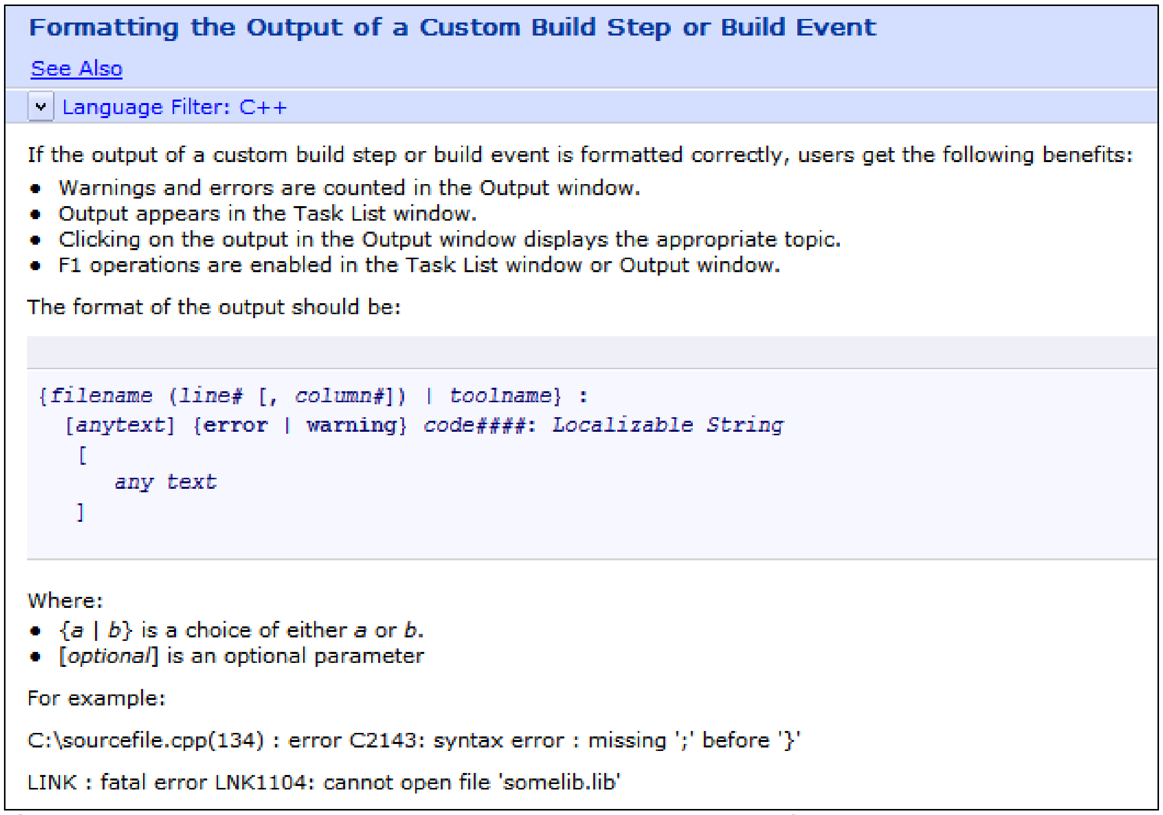
Object file has changed, reload?

Then I selected Yes, waited for the executable to be reloaded, and reran.

Appendix: Formatting GCC Diagnostic Messages to be Compatible with Visual C++

Visual C++ Diagnostic Message Format

The Visual C++ IDE requires the build output (e.g. *avr-gcc* diagnostic messages from the makefile build) to be formatted as the following article describes for it to successfully parse it.



Formatting the Output of a Custom Build Step or Build Event

See Also

Language Filter: C++

If the output of a custom build step or build event is formatted correctly, users get the following benefits:

- Warnings and errors are counted in the Output window.
- Output appears in the Task List window.
- Clicking on the output in the Output window displays the appropriate topic.
- F1 operations are enabled in the Task List window or Output window.

The format of the output should be:

```
{filename (line# [, column#]) | toolname} :
  [anytext] {error | warning} code####: Localizable String
   [
      any text
   ]
```

Where:

- {*a* | *b*} is a choice of either *a* or *b*.
- [*optional*] is an optional parameter

For example:

C:\sourcefile.cpp(134) : error C2143: syntax error : missing ';' before '}'

LINK : fatal error LNK1104: cannot open file 'somelib.lib'

[Source: http://msdn.microsoft.com/en-us/library/yxkt8b26%28v=VS.80%29.aspx]

GCC Diagnostic Message Format

I was unsuccessful at finding a definitive GCC (CP) Diagnostic message format specification in the online manuals, so I looked at the source code. The two snippets below were enough to figure out the apparent format is displayed as either of three forms, depending on calling conditions (presumably governed by compiler options):

```
("%s: %s",        progname, text)                   "p: dk: t"
("%s:%d: %s",     s.file, s.line, text)             "f:l: dk: t"
("%s:%d:%d: %s",  s.file, s.line, s.column, text)   "f:l:c: dk: t"
```

diagnostic.def:

```
/* DK_UNSPECIFIED must be first so it has a value of zero.  We never
   assign this kind to an actual diagnostic, we only use this in
   variables that can hold a kind, to mean they have yet to have a
   kind specified.  I.e. they're uninitialized.  Within the diagnostic
   machinery, this kind also means "don't change the existing kind",
   meaning "no change is specified".  */
DEFINE_DIAGNOSTIC_KIND (DK_UNSPECIFIED, "")

/* If a diagnostic is set to DK_IGNORED, it won't get reported at all.
   This is used by the diagnostic machinery when it wants to disable a
   diagnostic without disabling the option which causes it.  */
DEFINE_DIAGNOSTIC_KIND (DK_IGNORED, "")

/* The remainder are real diagnostic types.  */
DEFINE_DIAGNOSTIC_KIND (DK_FATAL, "fatal error: ")
DEFINE_DIAGNOSTIC_KIND (DK_ICE, "internal compiler error: ")
DEFINE_DIAGNOSTIC_KIND (DK_ERROR, "error: ")
DEFINE_DIAGNOSTIC_KIND (DK_SORRY, "sorry, unimplemented: ")
DEFINE_DIAGNOSTIC_KIND (DK_WARNING, "warning: ")
DEFINE_DIAGNOSTIC_KIND (DK_ANACHRONISM, "anachronism: ")
DEFINE_DIAGNOSTIC_KIND (DK_NOTE, "note: ")
DEFINE_DIAGNOSTIC_KIND (DK_DEBUG, "debug: ")
/* These two would be re-classified as DK_WARNING or DK_ERROR, so the
prefix does not matter.  */
DEFINE_DIAGNOSTIC_KIND (DK_PEDWARN, "pedwarn: ")
DEFINE_DIAGNOSTIC_KIND (DK_PERMERROR, "permerror: ")
```

[Source: http://gcc.gnu.org/viewcvs/trunk/gcc/diagnostic.def?revision=146533&view=co] diagnostic.c:

```
/* Return a malloc'd string describing a location.  The caller is
   responsible for freeing the memory.  */
char *
diagnostic_build_prefix (diagnostic_context *context,
                         diagnostic_info *diagnostic)
{
  static const char *const diagnostic_kind_text[] = {
#define DEFINE_DIAGNOSTIC_KIND(K, T) (T),
#include "diagnostic.def"
#undef DEFINE_DIAGNOSTIC_KIND
    "must-not-happen"
  };
  const char *text = _(diagnostic_kind_text[diagnostic->kind]);
  expanded_location s = expand_location (diagnostic->location);
  if (diagnostic->override_column)
    s.column = diagnostic->override_column;
  gcc_assert (diagnostic->kind < DK_LAST_DIAGNOSTIC_KIND);

  return
    (s.file == NULL
     ? build_message_string ("%s: %s", progname, text)
     : context->show_column
     ? build_message_string ("%s:%d:%d: %s", s.file, s.line, s.column, text)
     : build_message_string ("%s:%d: %s", s.file, s.line, text));
}
```

[Source: http://gcc.gnu.org/viewcvs/trunk/gcc/diagnostic.c?revision=166644&content-type=text%2Fplain&view=co]

Desired Translation from GCC format to Visual C++ format:

What essentially is required is to enclose line and column number information inside parentheses instead of colons as follows:

```
“p: dk: t”         ==>   “p: dk: t”
“f:l: dk: t”       ==>   “f(l): dk: t”
“f:l:c: dk: t”     ==>   “f(l,c): dk: t”
```

Translation implementation:

The implementation that I chose utilizes *sed*, a stream editor. The WinAVR tools send their output to *stderr*, while *sed* takes its input from *stdout*. Therefore, *stderr* must be redirected to *stdout* before *sed* is invoked. The Bourne shell redirection operator “`2>&1`” accomplishes the task.

Note that GNU *make* in WinAVR runs by default in the Bourne shell (*sh*), not the Windows command shell (*cmd.exe*).

The resulting makefile recipe command syntax is

avr-gcc gcc-options-and-parameters `2>&1 | sed -e "s_:\([0-9]*\):\([0-9]*\):_\(\1,\2\):_" -e "s_:\([0-9]*\):_\(\1\):_"`