VECTOR

VectorCAST Configuring Source and Include Paths

Version 1.0

2022-08-18

Application Note AN-ACT-1-009

Author Craig Pedersen

Restrictions Public Document

Abstract VectorCAST Configuring Source and Include Paths

Table of Contents

1 Overview

One of the more important configuration items when setting up VectorCAST is the organization of source and include paths. This application note will explain the different aspects of how these paths are used and configured when setting up test environments.

2 Use Cases

2.1 Use Case: Configuring Source and Include Paths

When building software outside of VectorCAST, setting up source and include paths is fairly straightforward. Using the build tool of choice (make file, IDE, or other), source code modules are added to the project and paths to the include files are specified as needed. If the paths are incorrect or incomplete, the builder or compiler will complain, and the solution is usually obvious. In VectorCAST however, there is a little more to it. Not only does VectorCAST need to know where files are located, it also needs to gather information for data types and stubbing purposes. Proper configuration of these paths will help insure a successful build. However, done incorrectly, problems are not always obvious and can lead to compilation, linker, or even execution errors. To help avoid those situations, this app note will provide guidelines on setting up the paths. Also, we will provide diagnostic techniques that can be used when trouble shooting any problems.

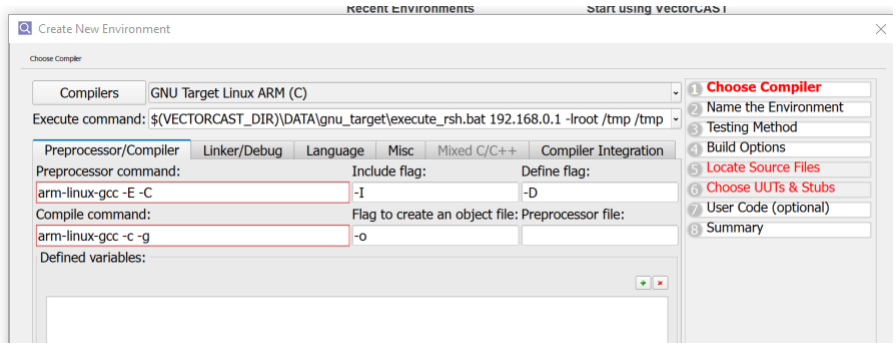
2.1.1 Getting Started

Before invoking VectorCAST, it is important to have the path of the build tools setup using the PATH environment variable (for Windows or Linux). By having the path set correctly, the compiler should know where the corresponding include files are located for that compiler. If you find yourself having to manually set include paths for the compiler inside of VectorCAST, then something is probably not setup before invoking VectorCAST.

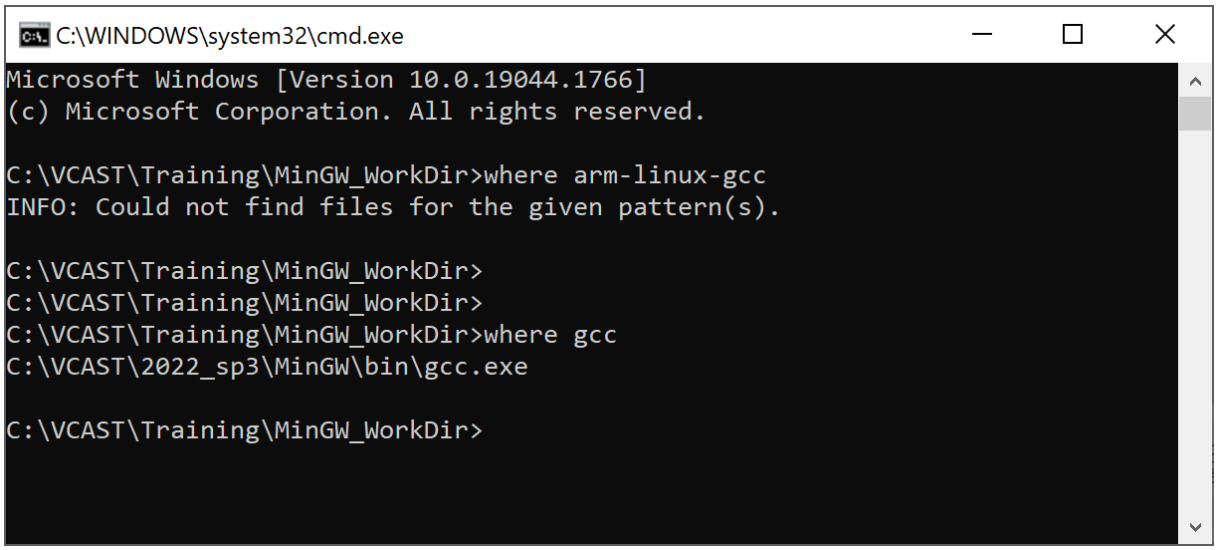
In addition to setting the PATH variable for the build tools, sometimes other setup is required. For example, to use the VisualStudio compiler & linker at the command line, a setup batch file is required. VectorCAST provides an example batch file in the installation under the “util” subdirectory (see run_vs_vcast.bat if you’re curious). On the other hand, the MinGW tool chain (which is a GNU compiler included with VectorCAST) does not require anything because VectorCAST is aware of that compiler and does the proper setup (even the PATH variable). Ironically, that can potentially cause problems if your target compiler is GNU based and is using “gcc” or “g++” for the compiler name. It’s possible you may pick up the MinGW compiler when something else is desired.

For other embedded compilers, various environment variables may be required as well. The Target Reference section of the RSP manual will have information on specific compilers.

When creating a new environment, if VectorCAST cannot find the build tools on the path, it will show “Choose Compiler” in red. In this example, the compiler name “arm-linux-gcc” is not on the path:



Regardless of the compiler, it is always good to verify the location of the build tools from VectorCAST's perspective. It's not uncommon to point to the wrong directory when more than one version of a compiler is installed. To verify, simply go to a shell prompt and use the "where" (Windows) or "which" (Linux) command to verify the version. Note that it is important to launch the shell from VectorCAST in order to inherit the same settings. To launch the shell, go to Tools->CustomTools->CommandPrompt. Here is a Windows example in both the good case and bad:



2.1.2 Configuring Paths in a VectorCAST Project

In VectorCAST, test environments can be created in a stand-alone manner, or they can be created inside a VectorCAST project. In stand-alone mode, the focus is on building just one environment and there may be no need or concern about path settings with respect to other environments. In a VectorCAST project however, that all changes. Ideally, you want path settings in one place that can be shared across one or more environments to eliminate duplication. A parallel example of this is a Visual Studio project or Eclipse Project where include paths are set at parent nodes and then shared by the source code below that node. The same is true in a VectorCAST Project.

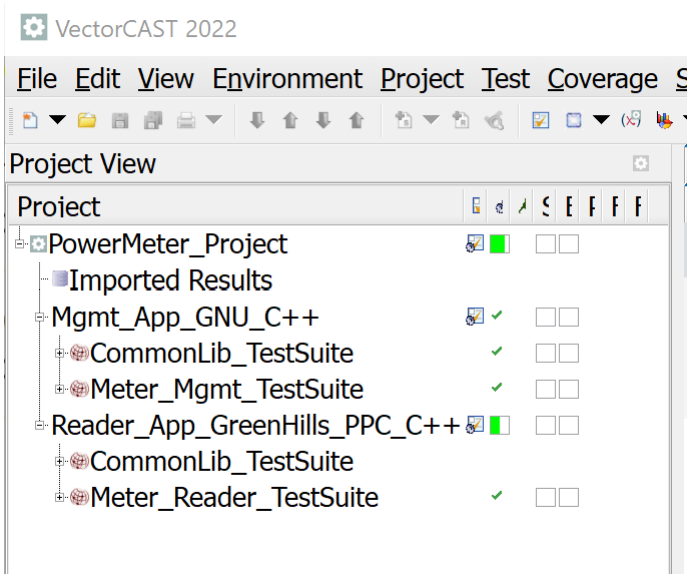
In general, include paths typically fall into one of the following categories:

- System– files that are used by one or more Applications
- Application – files that are unique to an application
- Shared– files unique to a component that might be shared across applications
- Module – files needed only by one module
- Compiler – files needed when using a specific compiler

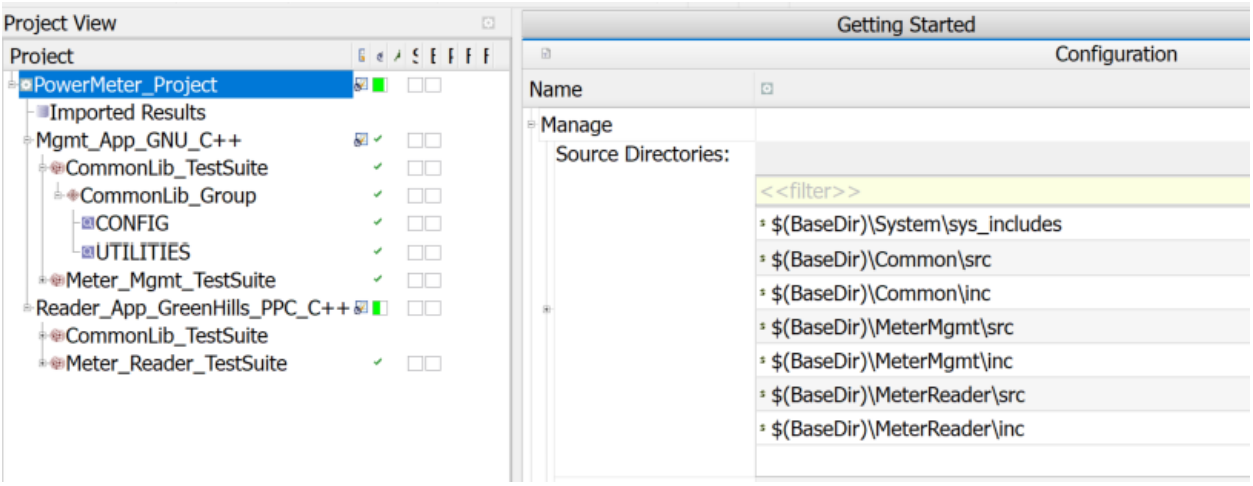
Conveniently, a VectorCAST project provides configuration nodes that support this type of organization as follows:

- Project Node – System Wide files
- Compiler or RSP Node – Files unique to a specific compiler or maybe an application
- TestSuite – Files shared across components or applications
- Environment – Files unique to all modules within that environment

In the screenshot below, we have a VectorCAST Project that contains test environments for a PowerMeter system comprised of two different executables: 1) A management application which monitors an array of Meter Readers and runs on a host computer using a GNU Compiler, and 2) An embedded Meter Reader application that runs on a PPC processor using a GreenHills compiler. The source code for each application is made up of a group of source code modules unique to the application and a common group of source code that is shared across both applications (note that the actual test environments are contained under the corresponding test suites but are not expanded here for brevity). Take a moment to examine the screenshot to understand the system organization. We will discuss how to configure the include paths below the screen shot.



The “proper” way to configure the include paths would be to set the System Wide include files at the top level of the project, and then the application and/or shared files at the Test Suite level. However, there are some practical benefits to configuring all source and include paths at the top level of project. Either approach can be used but for starters, we suggest setting the paths at the top level. If that doesn’t work for some reason, the paths can be moved down to their proper level. To change or add paths, just right-click on a Project node and select OpenConfiguration. The configuration at the top level might look like this:



We have now done enough configuration for VectorCAST to locate the source and include files and build the Unit Test Environments. However, there are other facets of path configuration that we will discuss in the next few sections. First though, we’ll do an overview of the Test Harness Build Process so we can understand the scope of the work.

2.1.3 Overview of the Test Harness Build Process

There are 2 main steps in the build process, a parse phase, and a compilation phase. The purpose of the parse phase is to understand the unit under test so that VectorCAST can create the test harness code that surrounds the unit. Once the test harness code is created, the next phase takes the generated code and compiles and links it into an executable. During this second phase, there is no special processing by VectorCAST. VectorCAST simply uses the provided build tools (typically, a compiler and linker) to create the executable.

For both phases, paths need to be set up to find the source and include files, just like you would in doing a system build outside of VectorCAST. During the parse phase, however, things are a little more complicated than the compilation phase. VectorCAST needs to understand what external entities need to be stubbed and type information is gathered for the data types used by the unit. We will now discuss each of these topics individually.

2.1.3.1 Stubbing

During the parse phase, VectorCAST discovers all the external entities that are accessed by the Unit Under Test (UUT). By default, VectorCAST will stub those entities if it finds a prototype definition. However, if a stubbed entity is also defined in a library that gets pulled in a link-time, there will then be a duplicate definition of that symbol. To avoid that situation, VectorCAST provides the ability to classify a path as a “library include” path. This indicates that for any prototype definition contained in an include file on that path, the entity should not be stubbed. The general rule is that the path(s) to the include files for a library should be classified as a “library include” path. The same can be said of Operating System’s functions.

2.1.3.2 Type Definitions

When writing test cases, it’s convenient to be able to populate and inspect any type definition. For example, if you have a type definition that is a “C” structure, you probably would want to access all its elements. In order to do that, VectorCAST creates what is referred to as “type handling” code for each data type. If the “type handling” code does not get generated for some reason, VectorCAST would not have access to the type. Contrary to that, there are times when it may be beneficial to have VectorCAST ignore type handling for certain directories. Library Paths (as described above in “Stubbing”) are an example of this. Usually, when you pull in a library, you don’t need the type information. There are 2 advantages of this: 1) the type handling code does not get generated on the target and therefore the test harness image is smaller (which may be desired on reduced memory targets), and 2) parse time is reduced when rebuilding an environment because VectorCAST does not spend time parsing all the files in the library paths.

There is another situation, though, that needs to be accounted for. That is the case when it is desirable to not stub library functions, but still have access to the type definitions. Let’s say, for example, a 3rd party library is pulled in and you want to use that library as-is. However, the Unit Under Test (UUT) uses some of the type definitions for global data or as arguments to UUT’s functions. In that case, you would want access to the type definitions. VectorCAST handles that situation by classifying the paths to the 3rd party library as “type-handled”. That means that the functions in the library are not stubbed but the types are still accessible.

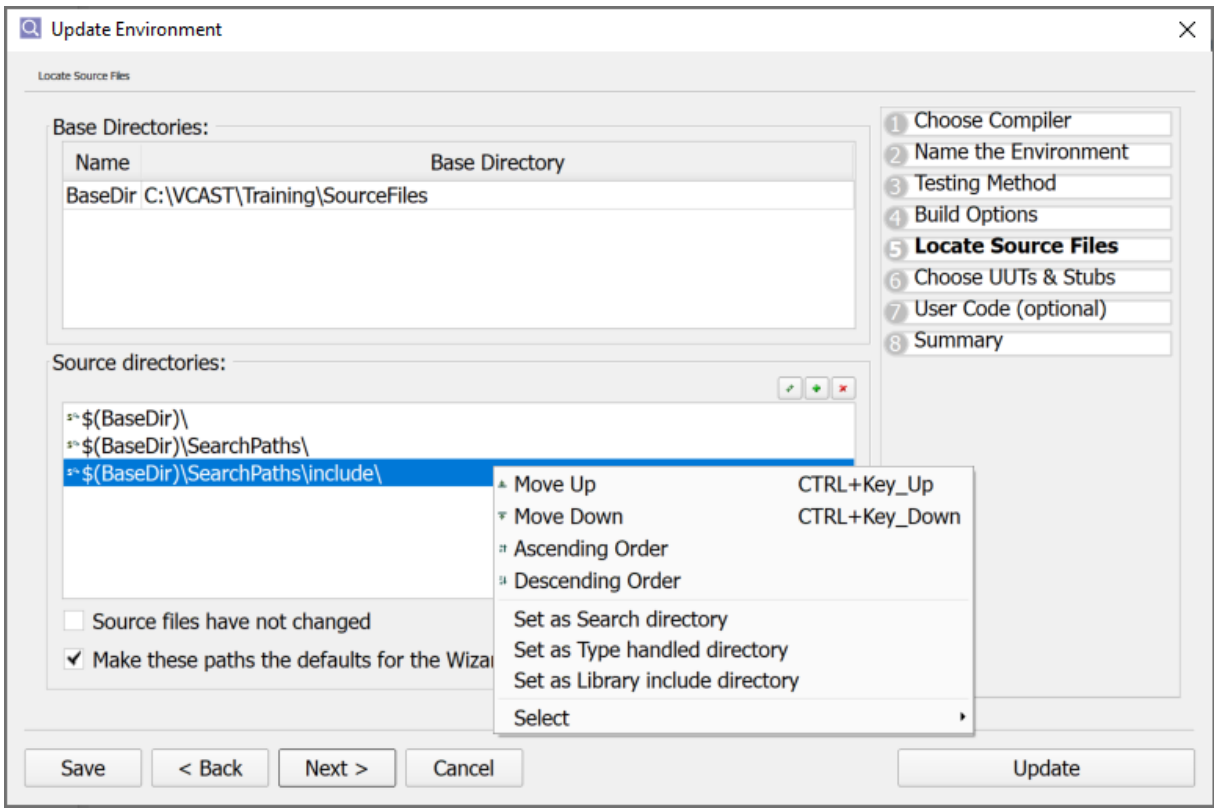
2.1.3.3 Summary of Path Classification

By default, VectorCAST treats all paths as “Search” directories. You can think of this as the “source” paths for the code under test. If the prototype for an external function is located in a “Search” path, the function will be stubbed. Also, all type definitions in “search” paths will be processed and will be available in the test case parameter tree. If a path is instead classified as “library include”, external functions that are defined on that path will not be stubbed and the type definitions will not be available. And finally, if a path is classified as “type handled”, the functions will still not be stubbed, but the type definitions will be processed and available. This table lists the various options:

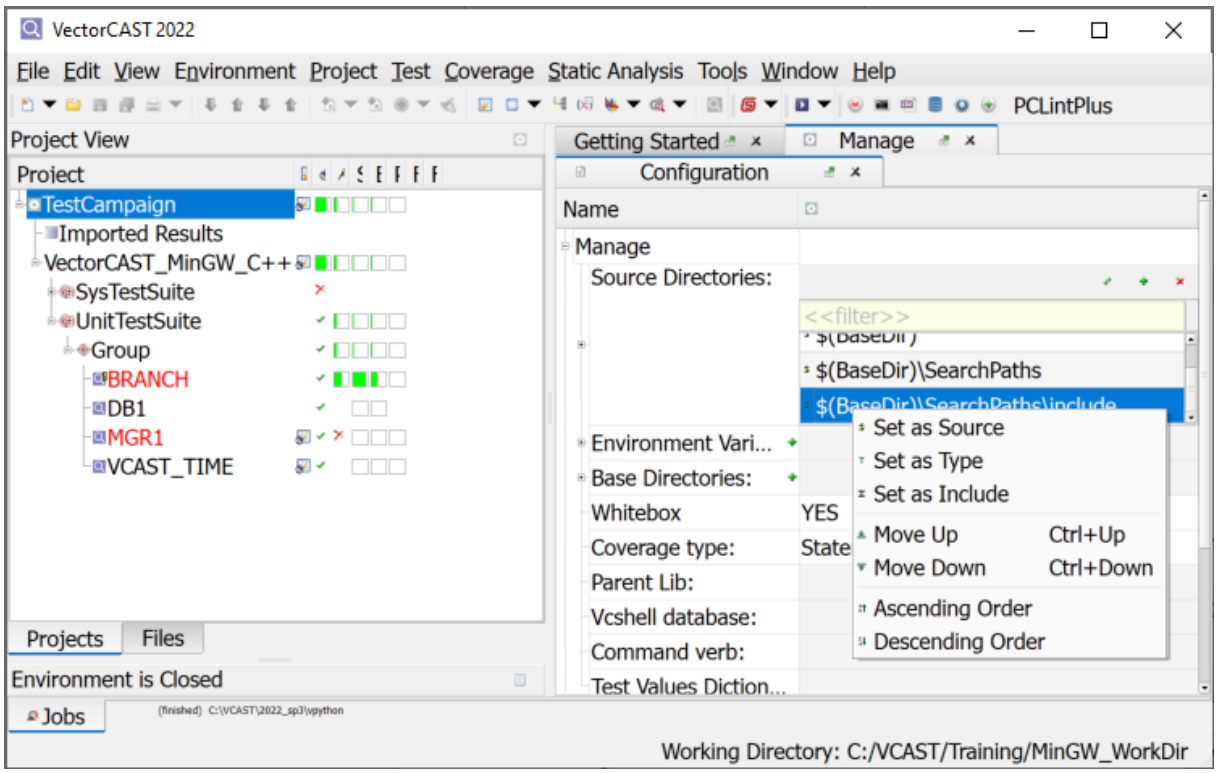
Path Type	Stubbed	Type Accessibility
Search	yes	yes
Library Include	no	no
Type Handled	no	yes

2.1.3.4 Setting the Path Classification

To change the pass classification, simply right mouse on the path and set the option to “Search”, “Type Handled”, or “Library Include”. This can be done at the project level or at the environment level when configuring the paths (as described above in the Configuration section). Here is an example of the options at the environment level:



The path settings can also be set or changed at the project level:



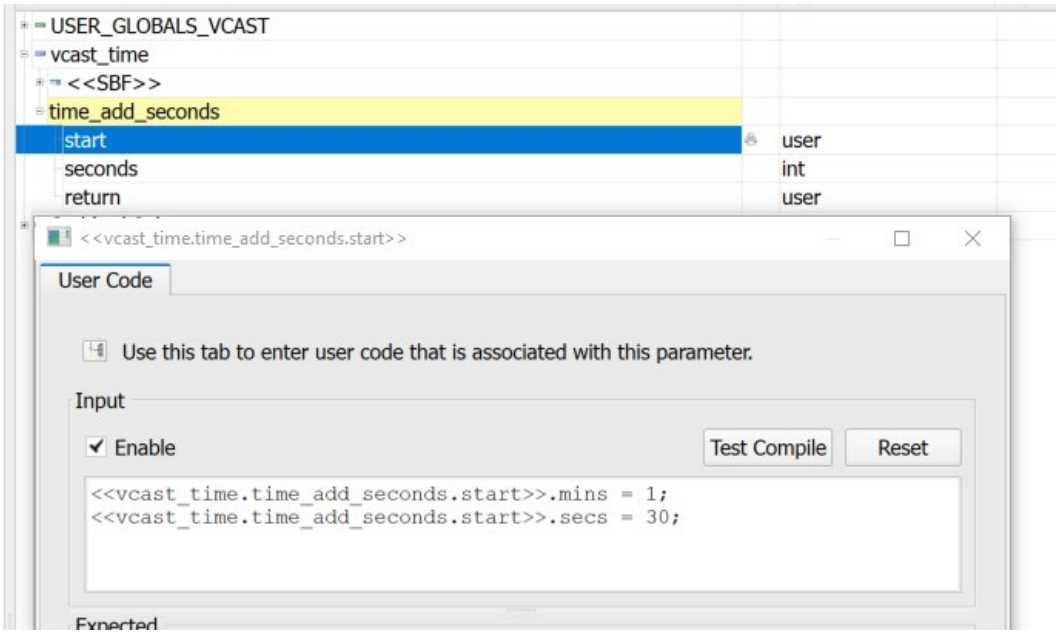
2.1.4 Using User Code for Type Handling

As explained above, if a type definition is defined in Library Include path, the typedef will not be available in the test case parameter tree when creating a test. The usual solution to that is to change the directory to “Type Handled”. However, there are certain situations where Type Handled directories can lead to build errors. Or it could be the case that the harness code is large, and the Type Handling Code needs to be reduced. Whatever the reason, it’s always possible to write user-code to get access to the types.

For example, let’s say we have a structure typedef that looks like this:

```
typedef struct time

{

int mins ;

int secs ;

} time_T ;
```

If this definition is in a Library Include path, it will show up as “user” in the parameter. VectorCAST will be aware of variables that are declared as “time_T” but will not be aware of the elements. To gain access, user-code is required. Here is an example where the input “start” timer is set to 1 minute and 30 seconds. Similar code could be written for the “return” values as well.



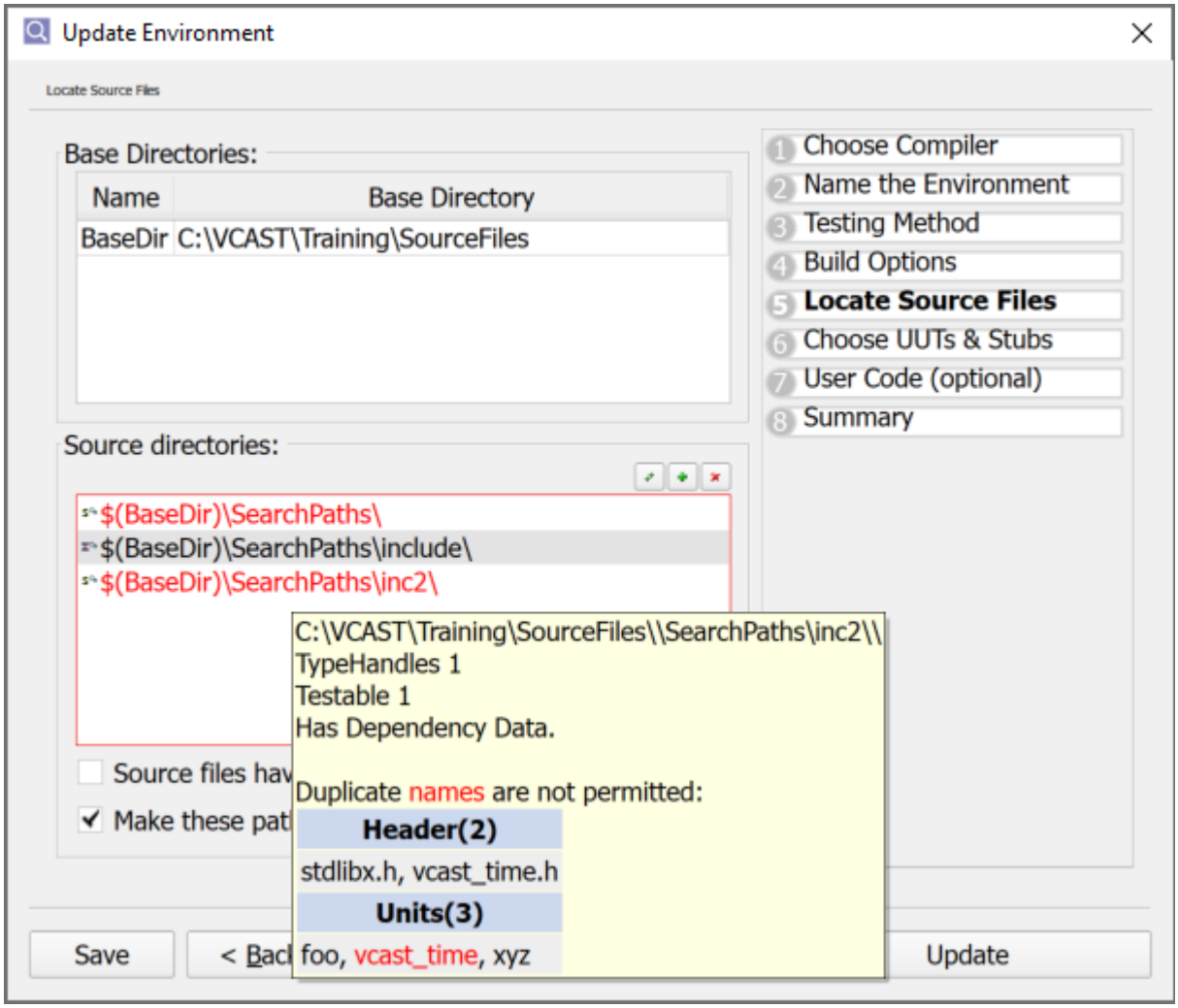
2.1.5 Diagnosing Pathing Errors

This section will discuss the diagnostic tools that are related to path configuration, in addition to a summary of various error scenarios.

2.1.5.5 Diagnostic Tools

2.1.5.5.1 File Path Inspection

One of the key tools used to diagnose pathing errors is the ability to hover over the path name to inspect the files located in that path. Also, any paths or files that are shown in red indicate a potential problem. If you suspect you might have a pathing problem, go to Environment->UpdateEnvironment and select step 5 (Locate Source Files) and hover over any of the paths to see the files located in the path. Here is an example where a rogue copy “vcast_time.c” was mistakenly placed in “inc2” and is shown in red:

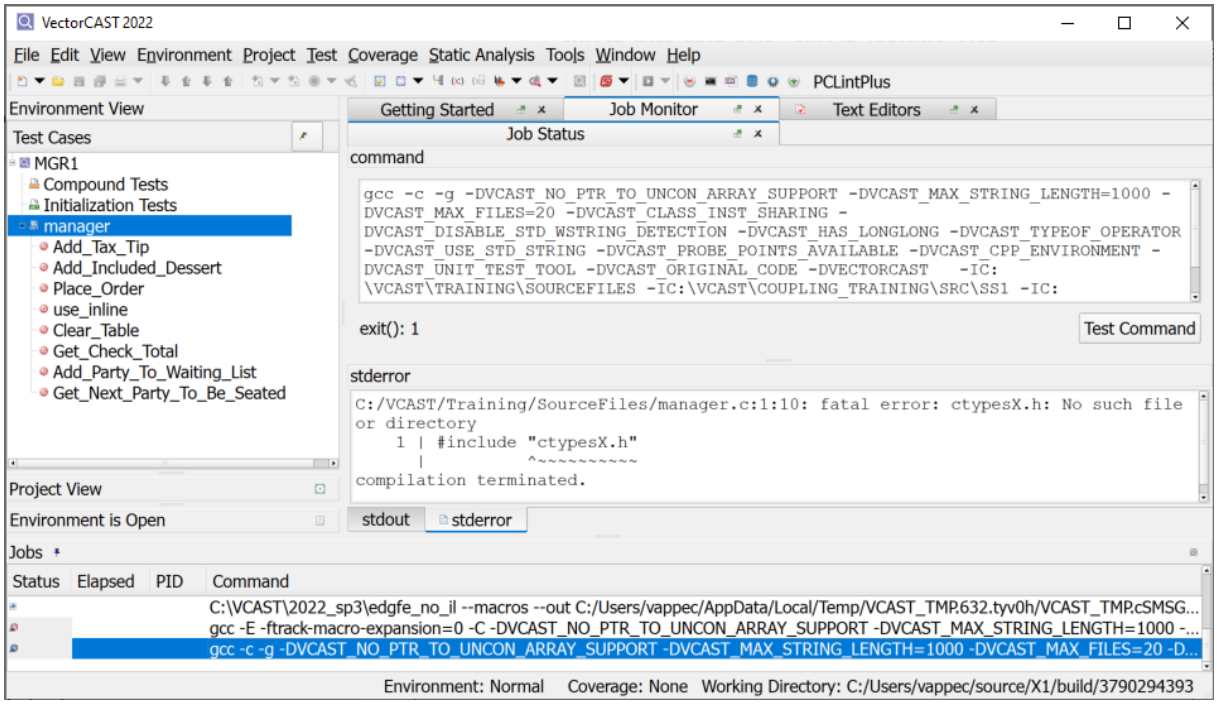


In general, anything shown in red should be addressed and cleaned up.

2.1.5.5.2 The Jobs Window

When an environment is built, VectorCAST invokes several commands that handle of most of the build details. Parsing, Compiling, Linking, etc... are some of the primary commands that VectorCAST uses at the command shell. If an error occurs, VectorCAST captures the error and displays it in the GUI. However, it may not be clear as to the exact cause of the problem and inspection of the command detail is helpful. In addition, that command can be copied from the Jobs Window and executed manually to replicate the issue. This can sometimes be a little quicker than going through the entire build process in VectorCAST.

In the example below, there is an error where the compiler cannot find an include file. The last line in the jobs window is usually the command associated with the error. If you double click on that line, you will see the command details along with the error that was printed to standard out. If you wanted to change a command option and do some experimentation with rerunning the command, you can simply copy the contents of the command window and paste it into a command shell (Tools->CustomTools->CommandPrompt).



2.1.5.5.3 Using Environment Variables for Include Paths

It is a common practice to use environment variables to access include paths. Let's say, for example, that the root of the source code repository for one engineer is located at "C:\JohnDoeRepo\Source". Other engineers in the group will probably have their source code in a different location on their workstation. To address that issue, an environment variable can be set to point to each worker's repository and then VectorCAST would use the environment variable instead of a hard-coded path. For example, the following Windows command would set up the SRC_BASE environment variable before launching VectorCAST:

```
> set SRC_BASE=C:\ JohnDoeRepo\Source
> %VECTORCAST_DIR%\vcastqt.exe
```

When using environment variables, sometimes problems occur that are not always obvious. To diagnose any suspected issues, it's best to use the command shell to evaluate the environment variable for correctness. This can be done by launching a command shell from VectorCAST (Tools->CustomTools->CommandPrompt) and evaluating the variable there. In Windows, for example:

```
> dir %SRC_BASE%
```

Note that VectorCAST uses the syntax “$(SRC_BASE)” to access environment variables regardless of using Windows or Linux. In VectorCAST, you might see something like this to access the include files in the repository:

2.1.6 Pathing Errors

2.1.6.6 Include File Not Found

When the compiler cannot find an include file, it will generate an appropriate error (e.g., “file not found”). Usually, the problem is straight forward, and the path just needs to be corrected or added to the list of include paths. Sometimes though, the problem is not always obvious, especially when sharing test environments across multiple users. For example, one user may set up a path to a local directory where other engineers don’t have access. Another common problem is if the source code specifies a subdirectory when including the file. For example, sometimes this type of construct is used in the source code:

- #include “sys/types.h”

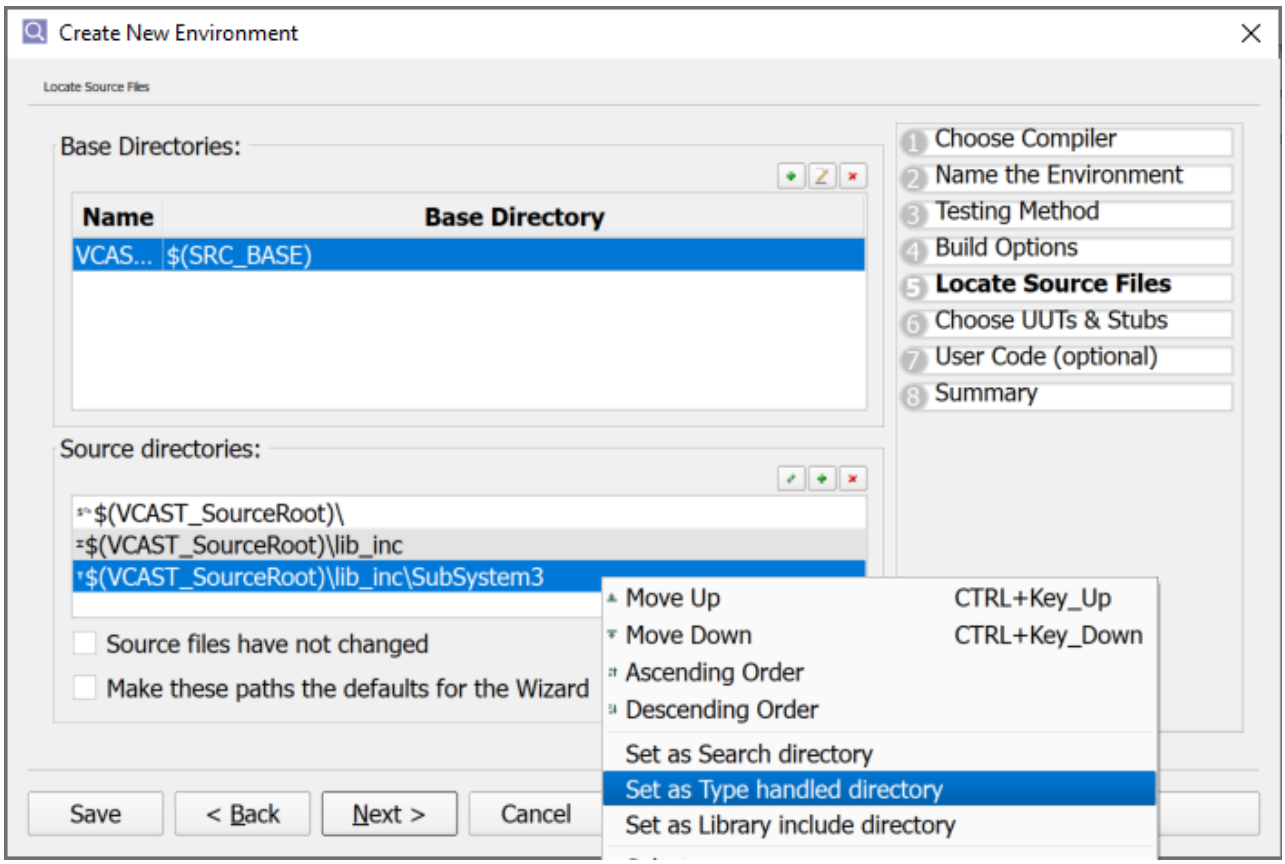
In this case, the path needs to point to the parent of the “sys” directory, and not the “sys” directory itself. This is one downfall of the “hover” capability as described above in the diagnostic tools. If you add a path to the “sys” directory and hover over it, you will see the “types.h” file. However, you still will get a compiler error if there is no path pointing to the parent.

2.1.6.7 Using Subdirectory Paths

There is another potential problem associated with configuring paths that include a subdirectory name. Let’s take an example of a 3rd party library where the include subdirectories are categorized by name and are stored under a parent directory called “lib_inc”. The source code that references the library might look like this:

- #include “SubSystem1/type1.h”
- #include “SubSystem2/type2.h”
- #include “SubSystem3/type3.h”

With this type of construct, you can specify just one path to parent directory (“lib_inc”) and everything will compile successfully. Because this is library code, you probably would specify the path as a Library Include Path. This means that no functions would be stubbed, and the type definitions would be suppressed. However, if you wanted exposure to the type definitions in just one of the directories (say “subsystem3”), you could add an additional path and classify it as Type Handled. This is not for the purposes of the compiler, but only for VectorCAST’s type handling process. The configuration would look like this:



2.1.6.8 Duplicate Include Files

This is one of the thornier problems to diagnose in any build system. Most compilers pick up the first occurrence of an include file and if the same include file name is located elsewhere, there will be no indication. One reason that this problem may occur in VectorCAST, as opposed to the normal system build, is the requirement to not include standard include directories. Some build environments have their own set of system include files (stdlib.h, etc...) that override the standard set provided by the compile. For example, with the GNU compiler, the option “-nostdinc” informs the compiler to not pull in the compiler’s system include files. If that option is accidentally not used in VectorCAST, then the wrong set of files would be pulled in. Another scenario where this problem can arise is if VectorCAST has a different ordering of include paths than what the normal system-build uses.

Regardless of the reason, pulling in the wrong copy of an include file could lead to a compiler or linker error, or possibly even an execution error. Unfortunately, the compiler nor VectorCAST has any built-in mechanism to detect this problem. It’s a problem that users need to be aware of and protect against by verifying that VectorCAST is using include paths in the same manner as a normal system build outside of VectorCAST.

3 Additional Resources

Some additional resources you may find helpful:

- VectorCAST/C++ User's Guide

4 Trademarks

All mentioned names are either registered or unregistered trademarks of their respective owners.

5 Contacts

For a full list with all Vector locations and addresses worldwide, please visit http://vector.com/contact/.