Renewable energy consultants

GL Garrad Hassan

BLADED EXTERNAL CONTROLLER USER MANUAL

Bladed External Controller User Manual

Garrad Hassan & Partners Ltd.
St Vincent’s Works, Silverthorne Lane, Bristol BS2 0QD England

www.gl-garradhassan.com

Contents

1 Overview

Bladed provides the facility for users to create and run their own controllers for the simulated turbine. These controllers can ask for simulation and model details, and 'control' the simulated turbine by setting various operational parameters. For example, the external controller could ask what the current angle of blade one is, and based on this value, set the demanded pitch actuator rate to try to achieve a different pitch.

The external controller has access to Bladed through a published API (Application Programming Interface). The Bladed API provides a suite of functions such as *GetNumberOfBlades* and *SetDemandedGeneratorTorque* so that an external routine – in this case a user-specified controller – can interact with the simulation. Documentation is available of the complete API in the accompanying *ExternalControllerAPI.chm*.

The mechanism by which the external controllers are integrated into the simulation is by compiling them as a *dynamic-linked library* (dll). These routines are compiled independently of Bladed itself, and are called by the simulation at discrete intervals to allow the controller to modify the turbine's controls. In order to function correctly, *dll*s must conform to a tight specification and be compiled correctly – which is what this manual is intended to help with.

2 Languages, Compilers and Development Environments

External controllers can be written in a variety of languages, but C, C++ and FORTRAN have the best support. Of these three, C++ is recommended as having the best combination of support and performance.

Dynamic-link libraries are specifically Microsoft Windows functionality – although analogous mechanisms exist on other platforms. Due to this, *dll*s have to be compiled using Microsoft libraries, and it is therefore far easier to use Microsoft's compilers and development environment – Visual Studio. Visual Studio 2010 is available to download from Microsoft at http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express.

There is a lot of support online for developing in Visual Studio 2010 and with C and C++ - for instance, at www.stackoverflow.com. The definitive standard C++ documentation is available online at http://www.cplusplus.com/reference/, which also hosts a useful forum that discusses common problems. There is a much more limited support network for FORTRAN, although some universities provide online tutorials and crib sheets.

This manual will concentrate on using Visual Studio 2010 to create C++ controllers. Examples are given of the equivalent controllers in Visual FORTRAN in Appendix A.

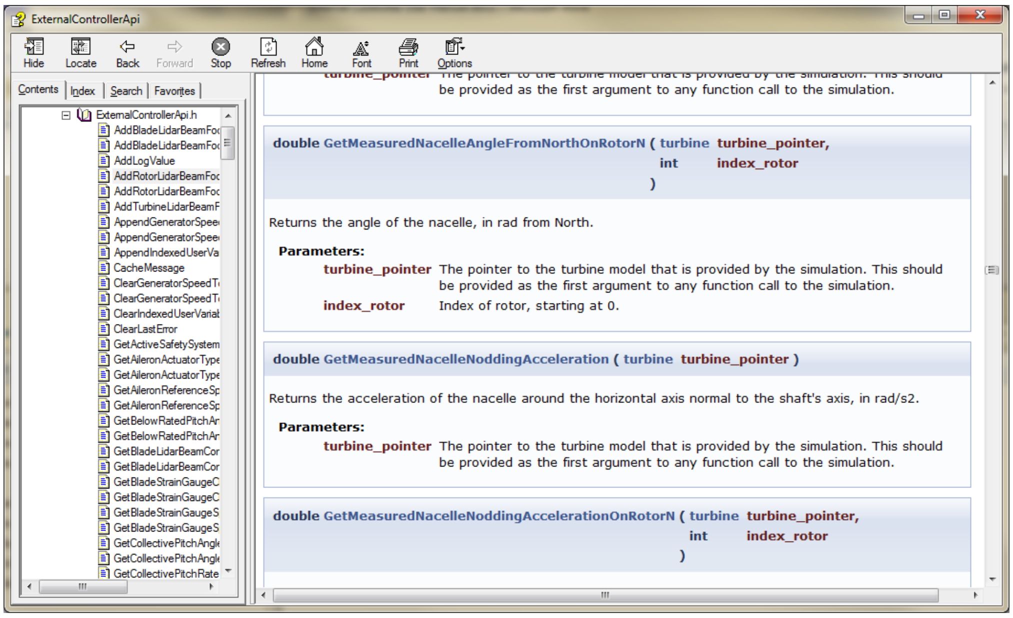
3 Setting Up A Simple Project

3.1 Prerequisites

Supplied with Bladed are a number of items which are required for creating external controllers.

ExternalControllerApi.lib	*A library required for the compilation of the external controller.*
ExternalControllerApi.h	*The definition of the available function calls that C and C++ code can make.*
mExternalControllerApi.f90	*A set of FORTRAN INTERFACE blocks that describe the function calls that FORTRAN code can make.*
ExternalControllerApi.chm	*A set of compiled HTML documentation of the API.*

The last is a comprehensive set of documentation of the commands that can be called by the external controller:

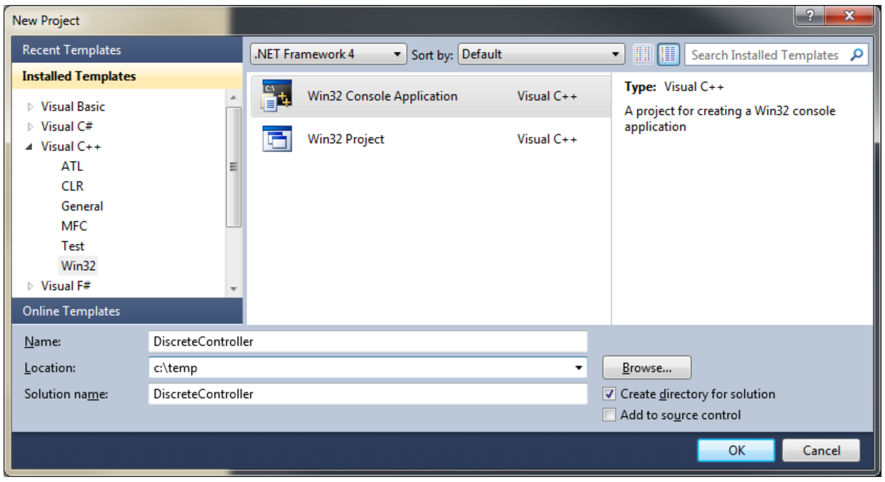


```
turbine_pointer  The pointer to the turbine model that is provided by the simulation. This should
                 be provided as the first argument to any function call to the simulation.

double GetMeasuredNacelleAngleFromNorthOnRotorN ( turbine  turbine_pointer,
                                                  int      index_rotor
                                                )

Returns the angle of the nacelle, in rad from North.

Parameters:
    turbine_pointer  The pointer to the turbine model that is provided by the simulation. This should
                     be provided as the first argument to any function call to the simulation.
    index_rotor      Index of rotor, starting at 0.

double GetMeasuredNacelleNoddingAcceleration ( turbine  turbine_pointer )

Returns the acceleration of the nacelle around the horizontal axis normal to the shaft's axis, in rad/s2.

Parameters:
    turbine_pointer  The pointer to the turbine model that is provided by the simulation. This should
                     be provided as the first argument to any function call to the simulation.

double GetMeasuredNacelleNoddingAccelerationOnRotorN ( turbine  turbine_pointer,
                                                       int      index_rotor
                                                     )
```

Although these are given in the C or C++ parlance, the basic data types can be read across into FORTRAN, and any further ambiguity can be resolved by looking in the mExternalControllerApi.f90 file which forms the FORTRAN interface.

3.2 Creating a Project in Visual Studio

In order to create a *dll*, we will need to set up a 'Project' in Visual Studio 2010. After opening Visual Studio, select File->New and then Project. This brings up a 'wizard' to configure the project:



Win32 Console Application	Visual C++
Win32 Project	Visual C++

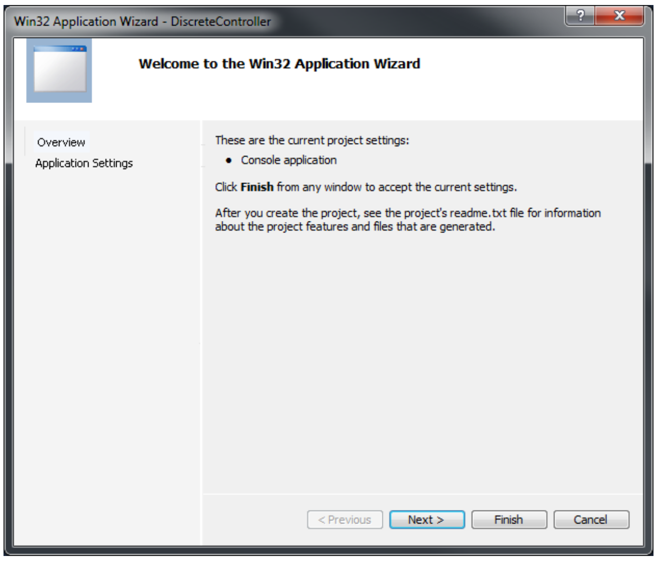
Type: Visual C++

A project for creating a Win32 console application

Name: DiscreteController
Location: c:\temp
Solution name: DiscreteController
[✓] Create directory for solution
[ ] Add to source control

Visual Studio will also create a 'Solution' to put the project into. This would be used to manage more than one project – for instance, if the controller's functionality were to be split into a number of libraries (*.lib files), each of which would be managed in their own projects. For the time being, we will create our controller called "DiscreteController.dll" in a single project and solution of the same name.

On the first screen of the project creation wizard, select "Win32 Console Application" from the Visual C++ templates, and provide names and a project folder. This will take you to the configuration screen:



Win32 Application Wizard - DiscreteController

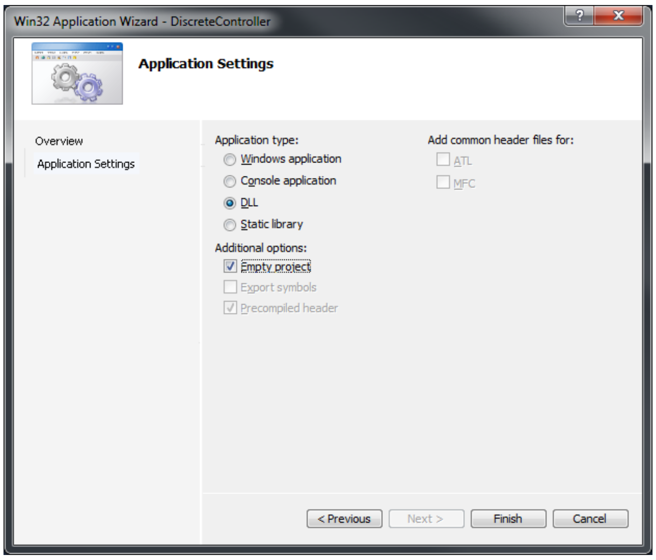
Welcome to the Win32 Application Wizard

These are the current project settings:
- Console application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

If you click 'Next', it will take you to the Application Settings screen:



Win32 Application Wizard - DiscreteController

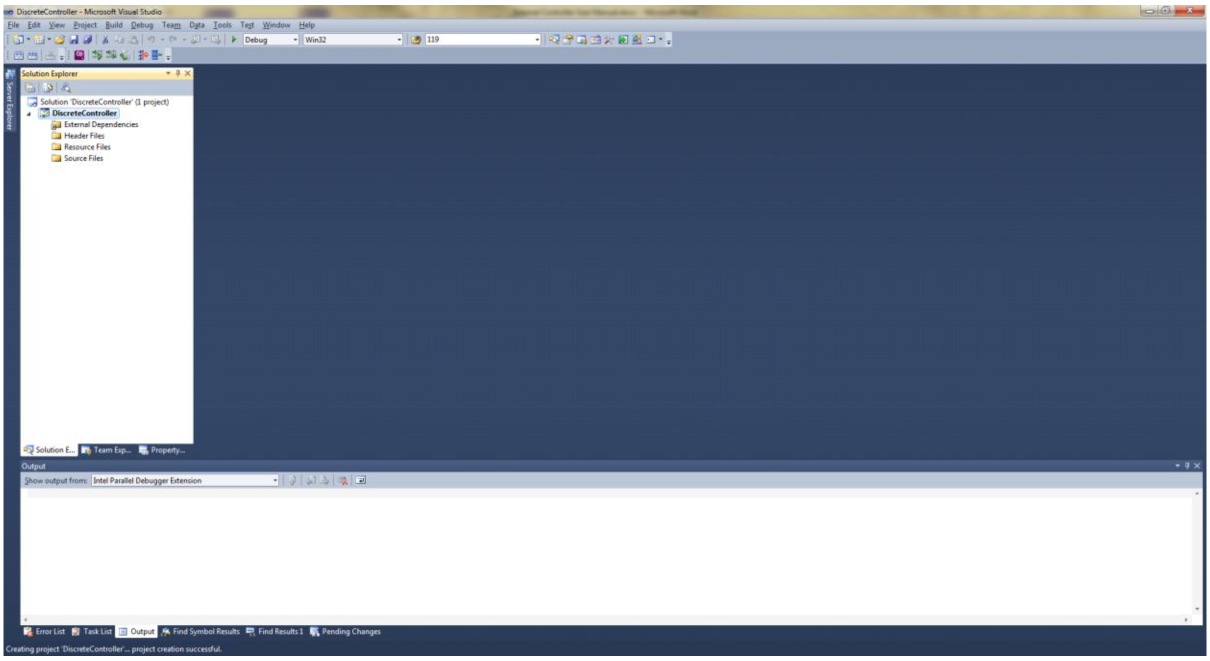
Application Settings

Application type:
- ○ Windows application
- ○ Console application
- ● DLL
- ○ Static library

Additional options:
- ☑ Empty project
- ☐ Export symbols
- ☑ Precompiled header

Add common header files for:
- ☐ ATL
- ☐ MFC

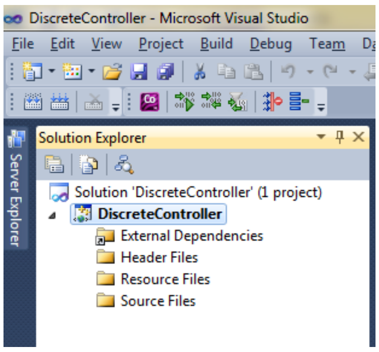
Select 'DLL' and tick 'Empty Project', then click 'Finish'. This will exit the wizard and leave you in the general Visual Studio development environment:


DiscreteController - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Win32
Solution Explorer
Solution 'DiscreteController' (1 project)
DiscreteController
External Dependencies
Header Files
Resource Files
Source Files
Output
Error List Task List Output Find Symbol Results Find Results 1 Pending Changes

Show output from: Intel Parallel Debugger Extension

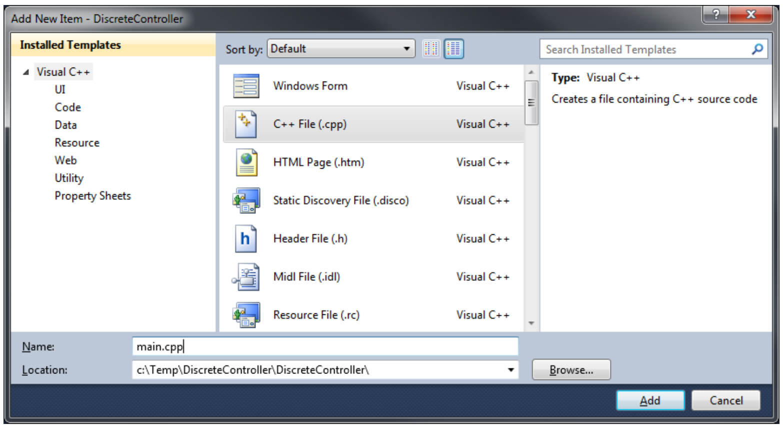
Creating project 'DiscreteController'... project creation successful.

The Visual Studio IDE (Integrated Development Environment) has a large number of 'Views' available, some of which are context dependent. The first view that we will need is the "Solution Explorer", which should appear by default upon creating a project or solution:


DiscreteController - Microsoft Visual Studio
File Edit View Project Build Debug Team
Server Explorer
Solution Explorer
Solution 'DiscreteController' (1 project)
DiscreteController
External Dependencies
Header Files
Resource Files
Source Files

From this window we can configure the project, add files, and compile the *dll*.

In order to be a valid project, we need to add an entry point to the application. Right-click on the "Source Files" or project node, and from the context menu select Add->New Item… This will bring up the "Add New Item" wizard:


Add New Item - DiscreteController
Installed Templates
Visual C++
UI
Code
Data
Resource
Web
Utility
Property Sheets
Search Installed Templates
Windows Form Visual C++
C++ File (.cpp) Visual C++
HTML Page (.htm) Visual C++
Static Discovery File (.disco) Visual C++
Header File (.h) Visual C++
Midl File (.idl) Visual C++
Resource File (.rc) Visual C++
Browse...
Add
Cancel

Sort by: Default

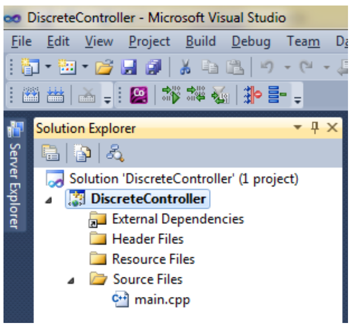
Type: Visual C++

Creates a file containing C++ source code

Name: main.cpp

Location: c:\Temp\DiscreteController\DiscreteController\

Select C++ (source) File, and name it “main.cpp”, which is the usual name for a C++ application entry point. This will add the file to your project under “Source Files”:


DiscreteController - Microsoft Visual Studio
File Edit View Project Build Debug Team
Server Explorer
Solution Explorer
Solution 'DiscreteController' (1 project)
DiscreteController
External Dependencies
Header Files
Resource Files
Source Files
main.cpp

3.3 Adding the Prerequisites

What we have up until this point is an empty VS project which is configured to create a dynamic-link library. We now need to configure the project so that it can control a Bladed and act as an external controller.

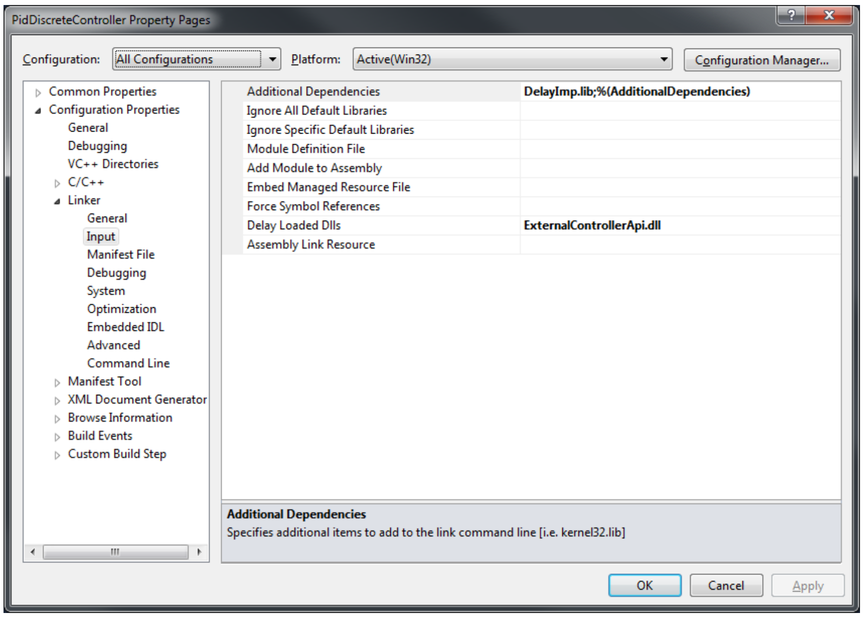
This section specifies how to set up the project for C and C++. For a set of equivalent instructions in FORTRAN, see “Appendix A Using Visual FORTAN”.

3.3.1 Linking to a Delay-Loaded DLL

The external controller of Bladed – such as we are creating here – does not speak to the Bladed executable (dtbladed.exe) directly, but calls functions that are contained in the ExternalControllerApi.dll, which you will find alongside dtbladed.exe in the Bladed distribution. If this dll is missing from the installation, an external controller cannot run.

In Windows-language, the external controller “delay loads” the ExternalControllerApi.dll. This means that the external controller is *aware* of the existence of the ExternalControllerApi.dll, but does not attempt to link to it until it is running within Bladed. This results in a smaller external controller (it doesn’t contain all of the functionality in ExternalControllerApi.dll) and it is forward compatible – so long as Bladed hasn’t removed any functionality that the external controller has relied on (this would only be in very unusual circumstances).

In order to delay-load a dll, we need to configure the project itself to expect it. Right-click on the project node in the Solutions Explorer and select “Properties”:


PidDiscreteController Property Pages

Configuration: All Configurations   Platform: Active(Win32)   Configuration Manager...

- Common Properties
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File
    - Debugging
    - System
    - Optimization
    - Embedded IDL
    - Advanced
    - Command Line
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step

Additional Dependencies	DelayImp.lib;%(AdditionalDependencies)
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	ExternalControllerApi.dll
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line [i.e. kernel32.lib]

OK   Cancel   Apply

In order to delay-load a dll, we need to include the VC++ library "DelayImp.lib", which is in your Windows installation. Visual Studio will already be configured to find standard Windows libraries, so it is not necessary to specify its folder in "Additional Library Directories" or its full path, as you would with a custom library.

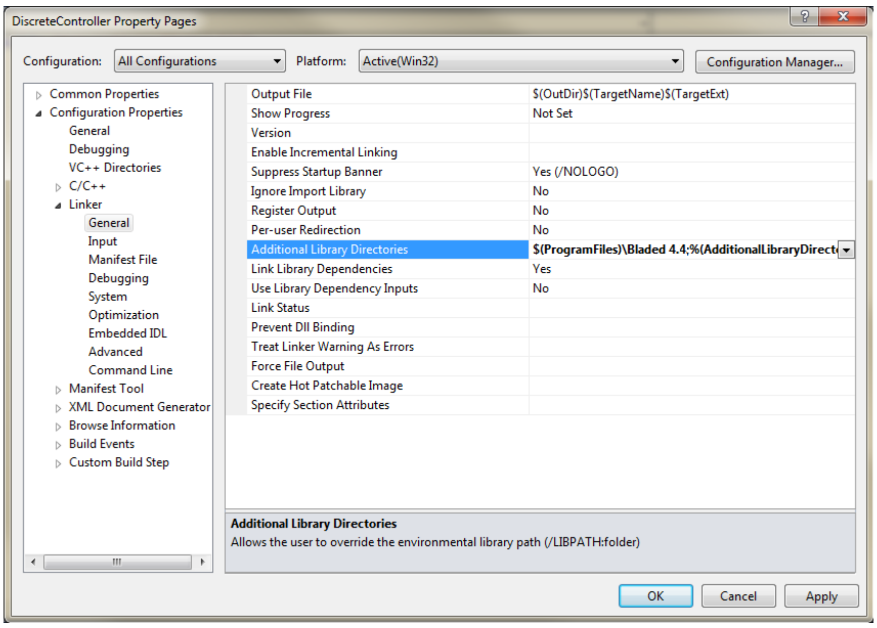
Visual Studio also allows you to specify the delay-loaded *dll*s, in the field "Delay Loaded Dlls", which is also on the Linker page.

***Tip**: Visual Studio contains independent registers of the settings for "Debug" build and "Release" builds. As you are likely to need to compile in Debug during development, and Release for the final version, it is best to use the same setting for both. You can do this by selecting "All Configurations" in the top-left of the properties screen <u>before</u> editing.*

3.3.2 Providing The Dll's Library File

The delay-loading mechanism requires access to the *dll*'s *lib* file, ExternalControllerApi.lib. This acts as a manifest of the available functionality, allowing the external controller to link to it during compilation. You can find *ExternalControllerApi.lib* in the Bladed installation directory, alongside the *ExternalControllerApi.dll*. The version of the *lib* does not need to be the same version as the *dll* – so long as the external controller only uses functionality that is available in the version of the *lib* that is available at compilation, and the version of the *dll* that is available at runtime. This makes Bladed external controllers forward compatible, so long as functionality is only added, not renamed or removed.

Specifying *ExternalControllerApi.dll* as a delay-loaded dll should automatically add a dependency on the *ExternalControllerApi.lib.* In the case of linker errors, however, it may be worth explicitly listing *ExternalControllerApi.lib* in the Additional Dependencies field on the Linker→Input page.


DiscreteController Property Pages

Configuration: All Configurations — Platform: Active(Win32) — Configuration Manager...

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**$(ProgramFiles)\Bladed 4.4;%(AdditionalLibraryDirect**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Additional Library Directories
Allows the user to override the environmental library path (/LIBPATH:folder)

***Tip**: Visual Studio has a number of pre-defined "macros" which allow many common locations to be 'soft-coded'. The macro $(ProgramFiles) will be replaced by the location of the Windows 'Program Files' directory on your computer – which may be different from your colleague's. It is also possible to create your own, for instance to a commonly used network drive.*

The last remaining step is to reference the lib file in the source. Double-click the main.cpp to edit it, and at the top of the file add:

```
#pragma comment(lib, "ExternalControllerApi.lib")
```

At this point, it should be possible to 'build' the project or solution (that is: compile the elements and link them together). Right-click on either the project or the solution, and select 'build' or 'rebuild' from the context menu – this should be completed without complaint. If you look in the output directory of your solution (by default, this is immediately under the solution directory and called either "Debug" or "Release", depending on your build settings) you should find a file called *DiscreteController.dll*.

3.3.3 Controlling the Turbine

The functionality that allows you to control the turbine within the Bladed simulation is now available to the dll, but if you tried to reference it the file would complain that it cannot find a definition of the functionality. The final step is to include the header file in the project.

In the source file, add the line:

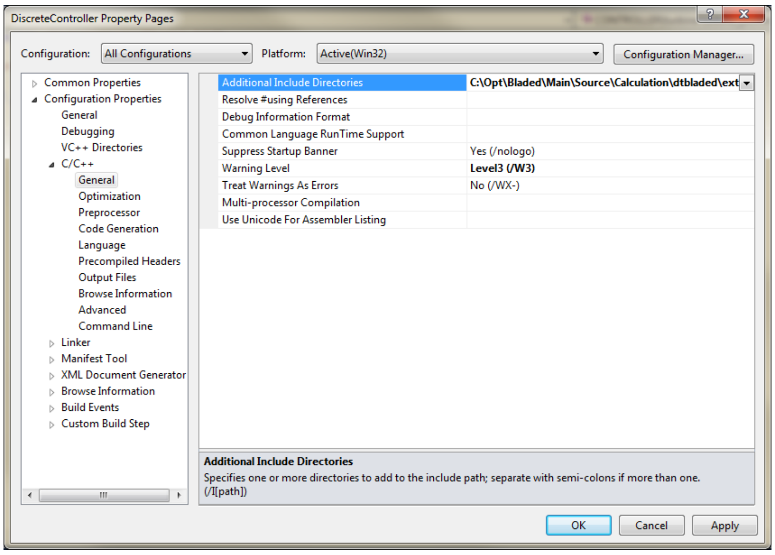
```
#include "ExternalControllerApi.h"
```

At this stage, this will fail to compile, and Visual Studio will underline the include in red, saying that it cannot find the file. There are two ways of remedying this:

1. Copy the header file into your project directory, or
2. Refer to the header file in its current location.

The choice between these two is a matter of policy, as the first will be more self-contained, but the second may be more easy to update if the version of Bladed is updated.

In the first case, nothing needs to be changed in the project's properties, but in the second, the location of the header file needs to be added to the "Additional Include Directories", on the C/C++→General page:



DiscreteController Property Pages

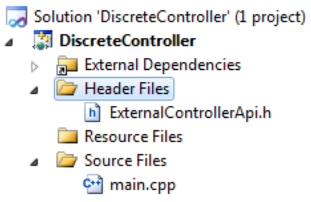
Configuration: All Configurations — Platform: Active(Win32) — Configuration Manager...

- Common Properties
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - Command Line
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step

Additional Include Directories	C:\Opt\Bladed\Main\Source\Calculation\dtbladed\ext
Resolve #using References	
Debug Information Format	
Common Language RunTime Support	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
Multi-processor Compilation	
Use Unicode For Assembler Listing	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

This will be enough to allow compilation, although it can be neater to also add this header to the project itself. If you right-click on the project, you can select Add→Add Existing Item... and navigate to the header file. This will then be added to the project under "Header Files":



- Solution 'DiscreteController' (1 project)
  - **DiscreteController**
    - External Dependencies
    - Header Files
      - ExternalControllerApi.h
    - Resource Files
    - Source Files
      - main.cpp

3.3.4 Summary

In summary, in order to successfully delay-load a dll, as we are doing here with ExternalControllerApi.dll, we need to:

- ☑ Include *DelayImp.lib* in the "Additional Dependencies" list.
- ☑ Specify *ExternalControllerApi.dll* in the "Delay Loaded Dlls" field.
- ☑ Add the location of *ExternalControllerApi.lib* to the "Additional Libraries Directories".
- ☑ Add `#pragma comment(lib, "ExternalControllerApi.lib")` to the source file.
- ☑ Add the location of *ExternalControllerApi.h* to the "Additional Include Directories".
- ☑ Add `#include "ExternalControllerApi.h"` to the source file.

We now have a dll that compiles correctly, but does not yet function as an external controller.

3.4 A Simple "Hello World" Controller

In order to constitute being a Bladed external controller – as opposed to an empty dll – we need to add a routine or routines for Bladed to run when required.

3.4.1 An Error-Free Controller

The most basic external controller which will compile and run without errors is:

```
#pragma comment(lib, "ExternalControllerApi.lib")

#include "ExternalControllerApi.h"    // This defines the C functions that can be called, as w
ell as 'turbine'.

using namespace GHTurbineInterface;

extern "C"
{
  int __declspec( dllexport ) __cdecl CONTROLLER (const turbine turbine_id)
  {
      return GH_DISCON_SUCCESS;
  }
}
```

This will do – quite literally – nothing.

***Tip**: The ExternalControllerApi.h defines a number of constants to aid clarity, including GH_DISCON_SUCCESS (which equals 0) and GH_DISCON_ERROR (which equals -1). Although not mandatory, using these can make the intent of the code clearer to another reader. For a full list, see ExternalControllerApi.chm* or the header file.

Tech-Tip: The first two lines have been explained already, but the rest might be of interest to some people:

`using namespace GHTurbineInterface;`	In C++, functionality can be split into 'namespaces' to avoid two functions of the same name being confused. All of the external controller functionality is contained in the *GHTurbineInterface* namespace. Namespaces are only a C++ concept, and this clause is not required for a controller compiled in pure C.
`extern "C"`	This clause is only required in C++ in order to tell the compiler to treat the function as a C-style function when exporting. Visual Studio will be compiling the project as C++ rather than C by default – although it is a project option.
`int`	This is the return type of the function. Bladed expects the external controller to return an integer upon completion: 0 for success; negative for an error; and positive for a warning.
`__declspec( dllexport )`	The tells the compiler to make this function visible from the dll. Any functions specified without this can only be called from *within* the dll.
`__cdecl`	This is the calling convention that the function is operating. __cdecl is the default for most C++ compilers, but specifying it explicitly makes certain and aids clarity.
`CONTROLLER`	This is the name of the function, which must be "CONTROLLER_INIT" or "CONTROLLER". If the first function is not found by Bladed, it will be skipped. If the second is missing, then Bladed will report an error and terminate.
`(const turbine turbine_id)`	This is the argument specification that Bladed will be sending to the external controller. The type 'turbine' is a typedef defined in *ExternalControllerApi.h*. The name of the argument could be anything, but you will need to pass it in as the first argument to every API function call. The keyword 'const' tells the compiler that the function should not modify its value.

3.4.2 Hello World – Using Bladed Messages

The next step is to make a function call:

```
#pragma comment(lib, "ExternalControllerApi.lib")

#include "ExternalControllerApi.h"    // This defines the C functions that can be called, as w
ell as 'turbine'.

using namespace GHTurbineInterface;

extern "C"
{
  int __declspec( dllexport ) __cdecl CONTROLLER (const turbine turbine_id)
  {
      ReportWarningMessage(turbine_id, "Hello World!");

      return GH_DISCON_SUCCESS;
  }
}
```

The Bladed API has a number of reporting functions which add messages into the Bladed reporting mechanism. The above line will result in "*** External Controller WARNING: Hello World!" being displayed on the screen and in the $ME message file. Other reporting messages are:

`ReportInfoMessage`	For messages to help the user – for instance, progress messages.
`ReportWarningMessage`	For warnings which allow the program to continue.
`ReportErrorMessage`	For messages to be printed before returning -1 as the error code.

3.5 More Advanced Controllers

From this point forward, writing more complex controllers just requires the use of the more complex API calls. For instance, calling `SetDemandedPitchAngle` will allow the controller to change the pitch angle on a specified blade, whereas `GetMeasuredTurbineLidarBeamFocalPointVelocity` allows the external controller to get the air speed determined by a specified Lidar station. Depending on the design of the turbine, only a subset of these API function calls may be relevant – for instance, not every wind turbine has a Lidar, and no tidal turbine does. Care should be taken not to attempt to execute irrelevant commands, as although they will not cause the program to crash it will also not complete the action – possibly obfuscating the intent of the controller.

3.5.1 General Use of API

ExternalControllerApi.chm provides a comprehensive list of all the functions available in the API, as well as all of the parameters or constant values that can be used for clarity. The documentation gives a short description of the item, and any necessary types and caveats. The list of functions is intended to represent the functionality that an external controller would have access to on a real turbine, and is therefore excluding some data that would be available to the simulation, but not to a genuine turbine operating in the field.

As a general guide, the functions can be split into five categories:

Simulation Functions	This covers things like the settling time and logging, and would not be meaningful on an operating turbine, such as

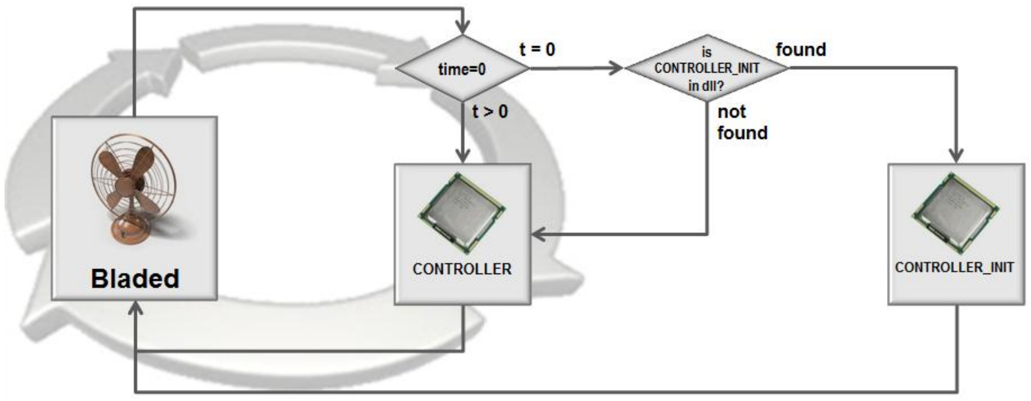
	GetCurrentTime().
Turbine Descriptors	This is read-only data describing the turbine, such as *GetNumberOfBlades()*, etc.
Nominal Values	This is a read-only data that is either referring to the initial state of the turbine (like the X, Y, Z positions of Lidars) or approximate inferred values (such as the flow speed at the rotor hub).
Measured Data	This is read-only data that would be coming from sensors on the turbine. In Bladed, transducer lag and signal noise is added to this data to imitate real-world behaviour.
Demanded Values	This is modifiable data used to control the turbine, such as by setting the generator torque.

All real numbers are given in double-precision (REAL(8) in FORTRAN). These *can* be implicitly or explicitly cast to single precision (floats in C, and reals in FORTRAN) although this would result in minor loss of fidelity, and warnings from some compilers.

3.5.2 Initialisation Using CONTROLLER_INIT

Bladed gives the facility to provide an initialisation controller, which will be run instead of the main controller on the first time-step. The syntax and functionality available is exactly the same as for the main controller, but it can be used to set initial conditions and states. It could also perform a number of actions before calling CONTROLLER itself.

The initialisation controller should be called “CONTROLLER_INIT” and placed in the same dll as the CONTROLLER. If Bladed can find CONTROLLER_INIT in the dll on the first time-step, it will run it in preference to CONTROLLER. If CONTROLLER_INIT is not present, or if the time-step is not the first, CONTROLLER will be run:



A file that provided both a CONTROLLER and a CONTROLLER_INIT would look something like:

```
#pragma comment(lib, "ExternalControllerApi.lib")

#include "ExternalControllerApi.h"    // This defines the C functions that can be called, as w
ell as 'turbine'.

using namespace GHTurbineInterface;

extern "C"
```

```
{
  int __declspec( dllexport ) __cdecl CONTROLLER (const turbine turbine_id);


  /*! Controller to be run on first timestep. */
  int __declspec( dllexport ) __cdecl CONTROLLER_INIT (const turbine turbine_id)
  {
      ReportInfoMessage(turbine_id, "Running CONTROLLER_INIT.");

      // Initialisation code…

      return CONTROLLER(turbine_id);    // Run CONTROLLER after setting up model (optional).
  }

  /*! Controller to be run on subsequent timesteps. */
  int __declspec( dllexport ) __cdecl CONTROLLER (const turbine turbine_id)
  {
      ReportInfoMessage(turbine_id, "Running CONTROLLER.");

      // Controller code…

      return GH_DISCON_SUCCESS;
  }
};
```

3.5.3 External Controller Return Value

The return type of the external controller is an integer, and this value reports to bladed the success or otherwise of the controller's execution.

Value	Meaning	Consequence
GH_DISCON_SUCCESS or 0	successful execution	Bladed will continue executing simulation.
integer > 0	execution with warnings	Bladed will report a warning message, quoting the return number.
GH_DISCON_ERROR or integer < 0	error during execution	Bladed will exit after having reported an error message to the $ME and $TE message files.

The failure to return an integer will generate compilation errors.

3.5.4 Error Behaviour

The Bladed external controller API does not generate exceptions under normal use, as neither C nor FORTRAN have an exception-handling framework. When an invalid command is executed, the controller is put into an error state which can be ascertained using `GetLastErrorCode(turbine_id)` and `GetLastErrorMessage(turbine_id)`. All 'Set' commands also return the error state to the controller; so `SetDemandedPitchAngle` would return GH_DISCON_SUCCESS (i.e. '0') on success, and GH_DISCON_ERROR ('-1') on failure.

'Get' commands can only return their result type. `GetLastErrorCode(turbine_id)` will indicate whether the 'Get' command has been successful, and should be used if the success of the operation is in doubt. Better yet is to check before making the query – for example, by asking whether there is an aileron before asking what its angle is.

When a 'Get' function fails, it returns a nominally invalid number. Functions that return doubles return *NaN* on error (for example, if the fourth blade's pitch was asked for on a three blade rotor). *NaN* (<u>N</u>ot <u>a</u> <u>N</u>umber) can be checked for, as it is the only number that doesn't equal itself: i.e.

NaN!=*NaN* is true, and some languages have a built-in *IsNan* or similar function. Unfortunately, not all data types have the equivalent of a *NaN* value, so values have been selected for other data types that should be clearly erroneous. These are available in the constants file as:

`GH_DISCON_ERROR_INT`	-666
`GH_DISCON_ERROR_DOUBLE`	NaN
`GH_DISCON_ERROR_FLOAT`	NaN
`GH_DISCON_ERROR_CHAR`	`"<function-returned-error>"`
`GH_DISCON_ERROR_STATUS`	-1

The API does not check for subtly erroneous values – for instance, values that might lead to a turbine over-speed. These values can only be determined after the simulation step has been performed, and control is returned to the external controller on the next time-step.

Appendix A Using Visual FORTAN

The version of FORTRAN used within Bladed is Intel Visual FORTRAN. This can be integrated moderately well into Visual Studio 2010, and the instructions here relate to Intel Visual FORTRAN 11.1B. It should be noted that the level of community support for FORTRAN of all types is far lower than for C or C++, and users who have no legacy constraints are recommended to use C++ to create their controllers.

A.1 Setting Up a FORTRAN Visual Studio Project

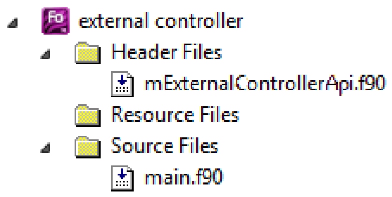
The basic creation of a FORTRAN project is the same as for C or C++, described in section 3.2. In the case of FORTRAN, no specific delay-loading settings need be added – it is only necessary to link to the library file.

A.1.1 Linking to the DLL

Add ExternalControllerApi.lib to the project's "Additional Dependences" in exactly the same way as for a C or C++ project, described above.

A.1.2 Including Bladed Functionality

Instead of a C header file, FORTRAN requires a library of INTERFACE blocks to define all of the functions in the API. These translate the arguments and return type from C to FORTRAN, and can be found along with the other external controller prerequisites. The file mExternalControllerApi.f90 must be added into the project – although its physical location is not important (i.e. it can reside anywhere on an accessible disc, and no additional references need be made).



```
external controller
    Header Files
        mExternalControllerApi.f90
    Resource Files
    Source Files
        main.f90
```

All of the INTERFACE blocks are contained in a single module, which can be included using normal FORTRAN syntax:

```
USE mExternalControllerApi
```

A.1.3 Summary

In summary, to use the Bladed API from FORTRAN:

- ☑ Add the location of *ExternalControllerApi.lib* to the "Additional Libraries Directories".
- ☑ Add *mExternalControllerApi.f90* to the project.
- ☑ Reference the interfaces module by using `USE mExternalControllerApi`.

A.2 A Simple "Hello World" Controller

In order to constitute being a Bladed external controller we need to add a routine or routines for Bladed to run when required. Bladed allows for two routines to be present in an external controller: "CONTROLLER_INIT" and "CONTROLLER". The first is run only on the first time-step (hence 'INIT'), whereas the second is run on every time-step – and on the first if "CONTROLLER_INIT" is not present.

A.2.1 An Error-Free Controller

The most basic external controller which will compile and run without errors is:

```
INTEGER(4) FUNCTION CONTROLLER [C] (turbine_id)

    USE mExternalControllerApi    ! This defines the C function interfaces that can be called

    IMPLICIT NONE

    !Compiler specific: Tell the complier that this routine is the entry point for the DLL
    !The next line is for the case of the Digital Visual Fortran compiler
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'CONTROLLER' :: CONTROLLER

    !Arguments
    INTEGER(C_SIZE_T), INTENT(IN) :: turbine_id[VALUE]

    CONTROLLER = GH_DISCON_SUCCESS

    RETURN
END
```

Like the C++ example, this controller will do nothing.

***Tip**: Like the C header file, the module* mExternalControllerApi *defines a number of parameters such as GH_DISCON_SUCCESS and GH_DISCON_ERROR. Although not mandatory, using these can make the intent of the code clearer to another reader. For a full list, see ExternalControllerApi.chm.*

Tech-Tip: The structure of the FORTRAN function is explained here:

`INTEGER(4) FUNCTION CONTROLLER`	The function declaration is to return an integer of 4-bytes – equivalent to a C int.
`[C]`	This specifies the calling convention as being 'C'-style (the alternative being STDCALL, the default). This is directly equivalent to __cdecl in C and C++.
`(turbine_id)`	Like the C routine, the FORTRAN controller accepts a single argument.
`!DEC$ ATTRIBUTES DLLEXPORT…`	FORTRAN identifies the DLL exports as a (commented) compiler flag.
`INTEGER(C_SIZE_T)… turbine_id`	The turbine_id is an integer, but of the size C_SIZE_T. This is defined in ISO_C_BINDING, and is guaranteed to be the size of a C++ pointer.
`CONTROLLER = GH_DISCON_SUCCESS`	The return from a function is set by equating a variable of the same name as the function to the return value.

A.2.2 Hello World – Using Bladed Messages

```
INTEGER(4) FUNCTION CONTROLLER [C] (turbine_id)

    USE mExternalControllerApi    ! This defines the C function interfaces that can be called

    IMPLICIT NONE

    !Compiler specific: Tell the complier that this routine is the entry point for the DLL
    !The next line is for the case of the Digital Visual Fortran compiler
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'CONTROLLER' :: CONTROLLER

    !Arguments
    INTEGER(C_SIZE_T), INTENT(IN) :: turbine_id[VALUE]

    !Variables
    INTEGER(4) RESULT

    RESULT = ReportWarningMessage(turbine_id, “Hello World”//CHAR(0))

    CONTROLLER = RESULT

    RETURN
END
```

This simple example also demonstrates the only two significant differences in the use of the API between C and FORTRAN:

1. In FORTRAN, you must always collect the return from a function call, and
2. When sending strings to the interface, a null character – `CHAR(0)` – *must* be appended to the string, or else risk segmentation faults.

A.3 Using the API Documentation

To ensure consistency and accuracy, the API documentation is auto-generated from the C code. This means that the arguments are described in C-notation. There is a one-to-one correlation between the C-types and their FORTRAN equivalent, and with familiarity the translation between them becomes automatic.

C Type	FORTRAN Type
int	INTEGER(4)
float	REAL
double	REAL(8)
char*	CHAR*
turbine	INTEGER(C_SIZE_T)

Note that nearly all of the API uses double precision for real numbers – that is double in C, and REAL(8) in FORTRAN. Although both languages can implicitly convert between single and double precision, it is recommended not to rely on this, as rounding errors can build up on each time-step.