Dictionaries and Components

DSix Dictionary Files

Description

This section provides an introduction to Dictionary files. The usage and syntax of Dictionary files will be discussed. Dictionary-header files and pragma-statements will be described. This section will show changes to ExternalIO.dic, changes to an existing component dictionary, and the creation of a new dictionary file.

What you will learn

Upon completion of this section, you will be able to:

- Understand the parts of a dictionary file along with their special features
- Use Visual Studio to modify existing dictionary files
- Use ExternalIO.dic to create new variables in a Dsix Project file
- Use dictionary header files to share data structures between components
- Use Visual Studio to add and build a new dictionary file

Anatomy of a Dictionary File

Dictionary files provide developers with a centralized location for declaring flight model parameters, either for the project interface (ExternalIO.dic) or at the component-level (e.g. AirData.dic).

Dictionaries have 4 mandatory features:

- A declaration of the component name: *Component AirData { }*
- An Input Structure: *Struct INP { }*
- An Output Structure: *Struct OUT { }*
- An Internals Structure: *Struct INT { }*

Anatomy of Dictionary Files - Continued

Component Declaration

The selected name needs to match the component's .dic, .cpp, and .h and will dictate the name of your associated component.

```
Component AirData {
    Struct INP {
                    } //end Inputs
    Struct OUT {
                    } //end Outputs
    Struct INT {
                    } //end Internals
} //end component
```

Anatomy of Dictionary Files - Continued

- Primary Structures (*INP, OUT, INT*)

These three structures define what parameters exist in the space controlled by the specific dictionary file.

NOTE: the primary structure name is “invisible” within a component, so you should proactively avoid duplicate naming

The number of substructures – and layers of substructures – that exist in each primary structure is at the discretion of the developer

Structure Examples – AirData.dic

- Substructures must be declared before primary structures

```
Component AirData {
        Struct INPUT_EOM {
                VECT3 UVWdotb { }
        }

        Struct INP {
                INPUT_EOM eom { }
        }

        Struct OUT { }
        Struct INT { }
}
```

Parameter would appear in AirData component as:

"eom.UVWdotb" – note lack of "INP" structure prefix

The Example Raises Questions!

What is "VECT3" and where did it come from?

Dictionary Headers provide a way to share commonly used datastructures across multiple dictionary files.

File type: ".dh"

Template/wizard-generated projects have a default dictionary header: **<ProjectName>_common.dh**

Dictionary Headers Explained

Open and examine **<ProjectName>_common.dh**

Note the differences in structure between a “dictionary header” (.dh file type) and a “dictionary file” (.dic file type).

One mandatory feature – a “structures { }” statement that encloses all declared structures

Note that the “common” dictionary header answer our earlier question as it contains a declaration for **“Struct VECT3 { }”**

```
Struct VECT3 {
    REAL x{ Description " " }
    REAL y { Description " " }
    REAL z{ Description " " }
}
```

Dictionary Headers Continued

The **"_common"** dictionary header is automatically included in all of the default dictionary files when a project is created using the wizard.

Numerous useful data types are included, and developers are encouraged to add their own commonly used data types to cut down on redundant declarations across multiple component dictionaries.

It is strongly recommended to include this dictionary header file in any user-generated dictionary files, as well:

"include <ProjectName>_common.dh"

Above line should be included at the top of the dictionary file.

A dictionary can include multiple dictionary header files.

Dictionary Special Features

Appear as "**pragma**" statements and allow BARDictionary to provide helpful features

- **D6ADDVAR20INT** generates a javascript file to add all ExternalIO.dic declared variables to the DSix Project
- **pack:8** explicitly defines 8-bit memory blocks for use with strict interface/network protocol
- **LATEXDOC** creates a companion **.tex** file that can be compiled in LaTeX to auto-generate an ICD

Using Dictionary Header

In the Wizard-generated flight model, note that ExternalIO.dic contains a structure called "**FANDM**".

This type of structure is a prime candidate for appearing in a **dictionary header** to avoid the need for redundant declaration across components.

Cut-and-paste the entirety of "Struct FANDM { }" from ExternalIO.dic into <Model>_common.dh

"**Rebuild**" the Dictionary project in the VS solution.

Note: changes to dictionary headers are NOT detected by VS and require a REBUILD rather than a BUILD

ExternalIO.dic and the DSix Interface

Using "**AddVarsExternalIO_20.js**" we can easily add large numbers of parameters to the DSix variable space while maintaining clear documentation of those parameters in **ExternalIO.dic**.

Example:

- Find "Struct PILOT"
- Add parameter to structure:

int sw_TEST { Description "Test switch for example" Convention "0 - OFF, 1 - ON" }

- COMPILE ExternalIO.dic
- Open DSix project in SAFE MODE
- On DSix command-line enter: ScriptRunFile("AddVarsExternalIO_20.js")
- SAVE the DSix project (this processes auto-code for the VS code project)
- BUILD SOLUTION in Visual Studio (this will compile all auto-code)
- In ComponentMapping.cpp – note that "m_pDep->Pilot_sw_TEST" is now a valid flight model variable

ExternalIO and the Interface - Continued

While that example seems laborious for a single parameter – note that this process can be executed for an arbitrarily large number of variables at once, representing a SIGNIFICANT time savings.

Additionally, this is the best way to “formally document” interface parameters if the pragma LATEXDOC statement is also used.

It is strongly recommended to place ANY DSix project variables that need to be added to a flight model project in the ExternalIO.dic, as it also makes it easier to recover “lost” variables in the event that they are inadvertently deleted from a project.

Making a New Dictionary

Easiest way – **copy** an existing .dic and **rename** the file and "Component" declarations to match the desired new component name

Example:

- Copy AirData.dic
- Rename to Electrical.dic
- In Visual Studio – right-click Dictionary project and
  - "Add->Existing Item -> Electrical.dic"
- Open Electrical.dic in VS and
  - rename the Component declaration
  - Delete the "typedefs" statement at the top of the file
  - Delete all structures EXCEPT FOR the required INP, OUT, and INT
  - SAVE and COMPILE the new dictionary file

Making a New Dictionary - Continued

Examine the <project>_FlightModel directory and note:

- 4 new files
  - Electrical.h
  - Electrical.cpp
  - Electrical.txt
  - Electrical.ndf

- The .ndf and .txt are typical files for compiling a dictionary (the .ndf acts as a header for the variable space declared in the dictionary)
- The .h and .cpp are new – they are auto-coded containers to be used to make a **NEW COMPONENT** that matches up with the new dictionary file that you just created

Adding a New Component

With these new files, we can now add a new component to the flight model project

Procedures are covered in detail on the following slides.

- Add the new files to the flight model project
- Create mapping functions (to get parameters from the interface into the component and vice versa)
- Add the component declarations to the Simulation Model
- Add all component initialization and destruction calls to SimulationModelInit
- Add the component to the core simulation loops

  (OnReset, OnStep, OnStop, OnIdle)

Adding a New Component - Continued

- In the Header Files filter of the FlightModel project (in VS)
  - Right-click – “Add -> Existing Item -> Electrical.h”
- In the Simulation Model filter of the FlightModel project
  - Right-click – “Add -> Existing Item -> Electrical.cpp”
- Open “Electrical.cpp” and note the default public functions
  - Initialize / OnReset / OnStop / OnIdle / OnStep
  - These are typical to the “SimComponent” class
  - We will need to wire these into the simulation loop for our component to be used by the flight model

Adding a New Component - Continued

First, we need to lay some groundwork

- At the end of ComponentMapping.cpp add:
  - void CSimulationModel::SetElectricalInput() { }
  - void CSimulationModel::GetElectricalOutput() { }
- In SimulationModel.h in the “ComponentMapping” section of the file add:
  - void SetElectricalInput();
  - void GetElectricalOutput();

- BUILD the FlightModel project

Adding a New Component - Continued

Next, we need to actually include our new component in the larger SimulationModel class:

- In SimulationModel.h
  - near the top add:
    - **#include "Electrical.h"**
  - In the protected section add:
    - **CElectrical*m_pElectrical;**

- In SimulationModelInit.cpp
  - Before AirData is called add:

```
m_pElectrical->Initialize();
GetElectricalOutput();
```

  - In ClearPointers() add
    - **m_pElectrical = NULL;**
  - In InitializePointers() add
    - **m_pElectrical = new CElectrical;**
  - In ReleasePointers() add
    - **delete m_pElectrical;**
  - In ZeroAll() add
    - **m_pElectrical->ZeroAll();**

  - **BUILD the FlightModel**

Adding a New Component - Continued

Now, we need to look at the Reset call orders:

- In SimulationModelReset.cpp
  - At the top of ResetNoTrim() and ResetWithTrim() add:

```
SetElectricalInput();
m_pElectrical->OnReset();
GetElectricalOutput();
```

Now, for the main simulation loop:

- In SimulationModel.cpp
  - In OnStep() after "GetTimingOutput() add:

```
SetElectricalInput();
m_pElectrical->OnStep();
GetElectricalOutput();
```

  - At the top of OnStop() add:

```
m_pElectrical->OnStop();
```

  - In OnIdle() after DSixRemoteGetData() add:

```
m_pElectrical->OnIdle();
```

BUILD the FlightModel

Adding a New Component - Continued

You now have a new component that can be used to model an aircraft subsystem, or act as a container for wiring in a Simulink-based model.

Reminders:

- It is up to YOU to determine the correct place in the simulation loop for your component – you need to understand the inter-dependencies between your SPECIFIC components in a flight model project
- Use the <component>.dic to contain and document as much of the variables used by a component as possible (i.e. avoid local declarations, both for clarity and performance)
- You need to use the ComponentMapping functions that we created early-on to transfer data between the interface and the component


DSIX
Flight Simulation Environment