

# **Software Standards and Tools for the NOvA Data Acquisition System Development Effort**

Kurt Biery, Glenn Cooper, Steve Foulkes, Gerald M. Guglielmo, Luciano Piccoli,  
Seangchan Ryu, Margaret Votava  
*Fermi National Accelerator Laboratory*

NOVA-doc-615

## *Abstract*

This document covers the software standards and tools required for the development of the Data Acquisition Software for the NOvA experiment. The main sections include: Coding Standards; Software Development Cycle; Code and Release Review Procedures; Software Languages and Environment; and Software Tools. The material represents a series of guidelines aid the developers in producing, testing, deploying and certifying a robust, high performance Data Acquisition System for the NOvA experiment.

# Table of Contents

1.....	Coding Standards	5
1.1.....	General Source Code Comments	5
1.2.....	Source Code Document Tags	5
2.....	Conventions and Formatting Styles	6
2.1.....	Naming Conventions	6
2.2.....	Formatting Style	9
2.3.....	Usage Rules	10
3.....	Software Development Cycle	11
3.1.....	Code Design	11
3.2.....	Code Development	12
3.3.....	Code Release Testing	12
3.4.....	Code, Document and Test Review	12
3.5.....	Code Release	13
4.....	Code and Release Review Procedures	13
4.1.....	Conditions for Triggering an Informal Review	13
4.2.....	Conditions for Triggering a Fagan Inspection	13
4.3.....	Informal Review	13
4.3.1.....	Informal Review Structure	14
4.3.2.....	Informal Review Procedures	14
4.4.....	Fagan Inspection	15
4.4.1.....	Fagan Inspection Structure	15
4.4.2.....	Fagan Inspection Procedures	16
5.....	Software Languages and Environment	18
5.1.....	Software Languages	18
5.1.1.....	Software Languages for Data Flow Applications	18
5.1.2.....	Software Languages for Display and Control	18
5.1.3.....	Software Languages for Small Tasks	18
5.1.4.....	Software Languages for Scripts	19
5.2.....	Software Environment	19
5.2.1.....	Operating Systems	19
5.2.2.....	User Interfaces to the Operating System	19
6.....	Documentation Standards	20
7.....	Naming conventions	20
7.1.....	Directory structure	20
7.1.1.....	C/C++	21
7.1.2.....	Python	22
7.1.3.....	Java	22
7.2.....	File Naming Convention	22
8.....	Software Tools	23
8.1.....	Compilers	23
8.1.1.....	C/C++ Compiler	23
8.1.2.....	Java Compiler	23

8.1.3	Python Compilers	23
8.2	GUI Tools	24
8.3	Authentication/security	24
8.4	Debuggers	24
8.4.1	C/C++ Debuggers	24
8.4.2	Java Debuggers	25
8.4.3	Python Debuggers	25
8.5	Build Systems	25
8.5.1	Build System for C/C++	25
8.5.2	Build System for Java	25
8.5.3	Standard make targets	25
8.6	Integrated Development Environments	26
8.7	Automated Documentation Generators	27
8.7.1	Automated Documentation Generator for Java	27
8.7.2	Automated Documentation Generator for C/C++	27
8.7.3	Automated Documentation Generator for Python	27
8.7.4	Languages without Automated Documentation Generation	29
8.8	Profilers	29
8.8.1	C/C++ Profilers	29
8.8.2	Java Profilers	29
8.8.3	Python Profilers	29
8.9	Source Code Checkers	30
8.9.1	C/C++ Code Checkers	30
8.9.2	Java Code Checkers	30
8.9.3	Python Code Checkers	30
8.10	Code and Build Requirements for Aiding Debugging Process	30
8.10.1	Trace Levels	31
8.11	Code Distribution	31
8.11.1	Issues with RPM for Multiple Version Support	31
8.11.2	UPS/UPD System for Code Distribution	32
8.12	Code Management Systems	33

# 1 Coding Standards

The Data Acquisition System software development effort will be a collaborative project. In order to provide quality control and make the code easily transferable among members of the team, the code needs to be developed in a consistent manner. Assuring consistency is made easier when automated tools can be applied to verify standards have been kept.

## 1.1 General Source Code Comments

Developers should include comments within sections of code to indicate general functionality and purpose at a minimum. They should keep in mind not only comments which will help them read and understand the code, but also what would be helpful to a new person trying to learn the code at a later date.

## 1.2 Source Code Document Tags

Where possible the use of Javadoc compatible document tags are to be used. Minimally this means in all Java, python, and C/C++ code. For python code and bash shell scripts this is also encouraged. While for those two languages there is not currently an application to parse the code and build documentation, it is possible some minimal level of this will be available and could be used.

The following table lists Javadoc tags that are required for C/C++ and Java code.

<i>Tag</i>	<i>Languages</i>	<i>Condition</i>
@author	Java, C/C++, python	Always (classes)
@version	Java, C/C++, python	Always (classes)
@param	Java, C/C++, python	As needed (methods)
@return	Java, C/C++, python	Always (methods)
@throws	Java, C/C++, python	As needed (methods)

All classes, functions and methods need to use tags. The version information goes within the block of code for the class. Methods should document all parameters, return values, and exceptions raised.

In general for C/C++ code, the main documentation for methods should go in the c/cpp files and not the include files. The documentation in the include files should cover the class documentation through the member fields using the documentation tags. However, for the methods a brief comment using the // comment format can be placed in the include file, but the tagged information should go in the c/cpp file.

Note that using the `/** */` format in the include file before the method declaration will cause the documentation parser to ignore the tags in the c/cpp file, so care must be taken to avoid that issue.

## 2 Conventions and Formatting Styles

There rules on naming conventions and formatting styles are specified in this section. Naming conventions cover rules for allowed characters and capitalization methods, classes, labels, etc. Formatting styles cover spacing of operands and labels and rules covering the general layout of the blocks of code in files.

### 2.1 *Naming Conventions*

The general rules on naming conventions have been pulled together from experience and also from reviewing various sources on standards from other projects. One place that served contributed to the rules selected was [http://www-cdf.fnal.gov/computing/coding\\_guidelines/CodingGuidelines.html](http://www-cdf.fnal.gov/computing/coding_guidelines/CodingGuidelines.html). Rules based on that reference are:

1. Classes may not have any public data members.
2. Class instance fields (instance fields) must have names beginning with an underscore character, followed by a lower case letter. The rest of the name should be a mix of upper and lower case letters. Upper case letters should be used to separate words in the name.
3. Static class fields (class fields) should be upper case letters with an underscore used to separate words in the name.
4. Local variables should start with a lower case letter, and be a mix of upper and lower case letters and digits. Upper case letters should be used to separate words in the name.
5. Class names should start with an upper case letter and be a mix of upper and lower case letters. Upper case letters should be used to separate words in the name.
6. Public method names should start with a lower case letter and be a mix of upper and lower case letters. Upper case letters should be used to separate words in the name.
7. Static method names should be named in the same manner as public method names (see above).
8. Namespaces should be all lower case letters with an underscore character used to separate words in the name.

The naming conventions for Java code are outlined in the following table. Note this is not the complete list of settings in the checkers, mainly just the naming

conventions. Additional rules will apply beyond naming conventions. For both the C/C++ and the Java checkers, the rules files to use will be provided. Note that regular expression patterns '\d' and '\w' mean 'digits (0-9)' and 'word characters (letters and digits)' respectively.

<i>Type</i>	<i>Rule</i>
Package Name Regex	[a-z][\w]*(\.[a-z][\w]*)*
Class Name Regex	[A-Z][\w\d]*
Abstract Class Name Regex	[A-Z][\w\d]*
Interface Name Regex	[A-Z][\w\d]*
Label Name Regex	[A-Z][A-Z0-9_]*
Private Instance Field	[_][a-z][\w\d]*
Package Instance Field	[_][a-z][\w\d]*
Protected Instance Field	[_][a-z][\w\d]*
Public Instance Field	[_][a-z][\w\d]*
Private Class Field	[A-Z][A-Z0-9_]*
Package Class Field	[A-Z][A-Z0-9_]*
Protected Class Field	[A-Z][A-Z0-9_]*
Public Class Field	[A-Z][A-Z0-9_]*
Private Final Class Field	[A-Z][A-Z0-9_]*
Package Final Class Field	[A-Z][A-Z0-9_]*
Protected Final Class Field	[A-Z][A-Z0-9_]*
Public Final Class Field	[A-Z][A-Z0-9_]*
Local Variable	[a-z][\w\d]*
Method Returning Boolean	[a-z][\w\d]*
Private Instance Method	[_][a-z][\w\d]*

<i>Type</i>	<i>Rule</i>
Package Instance Method	[_][a-z][\w\d]*
Protected Instance Method	[_][a-z][\w\d]*
Public Instance Method	[a-z][\w\d]*
Private Class Method	[a-z][\w\d]*
Package Class Method	[a-z][\w\d]*
Protected Class Method	[a-z][\w\d]*
Public Class Method	[a-z][\w\d]*
Private Final Method	[a-z][\w\d]*
Package Final Method	[a-z][\w\d]*
Protected Final Method	[a-z][\w\d]*
Public Final Method	[a-z][\w\d]*
Private Final Class Method	[a-z][\w\d]*
Package Final Class Method	[a-z][\w\d]*
Protected Final Class Method	[a-z][\w\d]*
Public Final Class Method	[a-z][\w\d]*
Parameter	[a-z][\w\d]*
Final Parameter	[a-z][\w\d]*

For C++, enumeration names and typedefs follow the same rule as the class name specified above. Enumeration fields and macros follow the label name regular expression rule defined above.

Similarly the naming conventions for Python scripts are outlined in the next table.

<i>Type</i>	<i>Rule</i>
-------------	-------------

<i>Type</i>	<i>Rule</i>
Module Name	[a-z_][a-z0-9_]*)([A-Z][a-zA-Z0-9]+
Class Name	[A-Z][a-zA-Z0-9]+
Function Name	[A-Z][a-zA-Z0-9_]*\$) ([a-z][a-zA-Z0-9_]*\$) ([_][A-Z][a-zA-Z0-9]*
Method Name	[A-Z][a-zA-Z0-9_]*\$) ([a-z][a-zA-Z0-9_]*\$) ([_][A-Z][a-zA-Z0-9_]*\$) ([_]{2}[a-z]+[_]{2}
Argument Name	[a-z][a-zA-Z0-9]*
Variable Name	[a-z][a-zA-Z0-9]*

A similar naming convention should be adopted for C/C++ and bash shell scripting.

## 2.2 **Formatting Style**

General formatting conventions for Java code are outlined in the following table. Note this is not the complete list of settings in the checkers. Again for both the C/C++ and the Java checkers, the rules files to use will be provided.

<i>Type</i>	<i>Rule</i>
Space After Statement Keyword	TRUE
Line Length	120 (guideline of 80)
Space Around Binary Expression	TRUE

From the JCS rule editor application, the following explanations for a few rules are provided for clarification:

- “Space After Statement Keyword”

This rule specifies whether you have a space ' ' after a statement keyword. Statements are if, while, for, ...

```
if (a < MAX) {
    for (int i = 0; i < MIN; i++) {
        ...
    }
}
```

```

    }
}

```

- “Space Around Binary Expressions”

Binary expressions have operands on both sides of the operator.

```

int i = MAX * 2;
long l = i + (MIN - MAX / 2);
for (int i = 0; i < 10; i++) {
...
}

```

Another formatting rule is the use of four (4) spaces instead of tabs for indentation where feasible. This rule should apply to all C/C++, Java, and Python code and Bash shell scripts. One exception to the rule is makefiles which require some lines to begin with a tab.

There is also a rule specifying that an open bracket should appear on the header line for the block, with the close bracket line on a separate line. An example of this rule is shown here:

```

public JPanel getPanel() {
    return thePanel;
}

```

## 2.3 Usage Rules

There are many usage rules spelled out in the literature on coding excellence. In the case of C++ see for example *Effective C++* and *More Effective C++* by Scott Myers (Addison-Wesley), or *C++ Coding Standards, 101 Rules, Guidelines, and Best Practices* by Herb Sutter and Andrei Alexandrescu. These rules will not be restated here as reference books are devoted to properly describing them and which ones are relevant may not be completely known ahead of the development cycle. However some of these guidelines may be adopted in the future and applied to the code, especially if it improves the quality of the code. Usage rules can evolve over time, here we only document an initial baseline as the foundation for the future.

Here are some usage rules as a starting point:

1. Referencing static final fields. This covers a Java rule on static final fields. If we have a class called AClass with a static final string named EXIT, when referring to this string in a method of BClass, one should use the class name and not an instance name. In other words one should refer to AClass.EXIT and not myLabel.EXIT where myLabel is an instance of AClass.

2. Static class methods over functions. This rule refers to C++ use of functions. Use of global functions should be avoided. Instead, static class methods should be used in place of functions. It is recognized this only applies to code being developed as we have no control over libraries and APIs provided by third parties. It is noted that finding static class methods in a large body of code is easier than global functions as the class name quickly narrows the search. It also allows for better grouping of related functionality, as multiple container classes can be defined for specific kinds of methods, than separate functions possibly scattered throughout the code base.
3. For C++ no classes should be defined in the global name space. For Java no classes should be declared outside of a package.
4. Header files may not contain any using directives or declarations.
5. Do not put any conditional compilation clauses which define new member datum or change size of any object based on value of a locally defined variable in a header file.
6. Do not use C-style casts. Instead use the appropriate C++ `dynamic_cast`, `reinterpret_cast`, or `const_cast`.

### **3 Software Development Cycle**

There is a larger outer development cycle driven by preparing and executing major releases of the software. That cycle is broken down into four main phases: design and development for the release; testing for the release; review of development and testing; release of code. Within each main phase there are smaller cycles which form can be iterated until the phase is complete if necessary. The structure of the cycle allows assessment of status toward the release and the ability to insure consistency across many sub-projects for the Data Acquisition System.

#### **3.1 Code Design**

During the design phase, developers are creating a blueprint for building the software to meet the requirements defined by the requirements document. This phase is a very important step in the process of delivering performant, reliable code in a timely manner. Careful attention during the design phase minimizes the risk of having to re-write large portions of the code. This phase will produce a design diagram in UML (or similar notation) along with a text document describing the diagram. Once the design has been completed, there will be an informal review. Feedback from the review will be used to update the design diagram and documentation as appropriate and the design will then be available to start the coding cycle.

### **3.2 Code Development**

During the code development phase, developers are performing a series of activities leading to the conclusion of this phase with software that: provides functionality requested; adheres to standards; has appropriate tests built; and is well documented. There may be multiple iterations of writing the code and test code for a new feature, running these new tests, and documenting the new feature. The number of iterations is coupled to how many feature are needed for the release. The developer may request an informal review or Fagan inspection if they have strong concerns about some section of the code or a design choice. Once the developer is satisfied with the code, the tests and documentation have been properly updated, and the software is packaged in the same format as will be used in the production system, then this software is ready for the next phase.

### **3.3 Code Release Testing**

This phase does not begin until the design and development phase is complete. In this phase formal testing is done and all tests are run for the software package being developed. This differs from the testing in the development phase where the developer may chose to only run the tests specifically for the new functionality. Another difference is this phase requires documentation of the test results in a manner that can be presented for review. In the development phase, the running of tests are primarily to aid the developer in the process of creating the code and do not need to be documented for others to see.

All major releases of software are required to be tested on the integration test facility before being released into production. All minor releases are strongly encouraged to pass through the integration facility as well.

If a problem is uncovered in this phase, then the code needs to fixed and this phase starts from the beginning once the fix is complete. The phase ends when the testing has been successful and documentation including the output, with timestamps, from all tests has been produced and is ready for review.

### **3.4 Code, Document and Test Review**

During this phase a set of reviewers, differing for formal and informal reviews, are provided the software documentation, test results and source code for the package under review. The structure of the reviews may differ between formal and informal, but the goal remains the same in both cases. This goal is to insure as much as possible that the software is ready for release on the basis of performance, reliability and documentation. The reviewers can request clarification on any issues they find and can request changes to the code or documentation. Any changes requested to the software would push the process back to the code release testing phase, whereas requests to change documentation would require only a review of

the updated documentation. At the end of this phase the software is ready to be released.

### **3.5 Code Release**

During this phase the software as packaged prior to the testing phase is placed in an official distribution repository and announced as released. This is the shortest phase of the main cycle.

## **4 Code and Release Review Procedures**

The goals of the code review process are to insure robust, reliable and well documented software and to expose developers to the ideas and techniques of other developers. Both of these goals serve to improve the overall quality of the software produced. For the first goal it is fairly easy to see how the software quality is improved by the process. Achieving the second goal raises quality by improving the breadth and experience of the people developing the code.

There are two types of reviews defined for this project, an informal review and a Fagan inspection. The former review is more of an internal check that everything that needs to be done has been completed properly. The latter review is more structured and focused in nature. At least one member of the review board is expected to be not on the development team.

### **4.1 Conditions for Triggering an Informal Review**

An informal review can be requested by a developer if they have concerns about a specific section of code or issues with meeting a specific requirement. There should be an informal review before any code release as part of the normal development cycle.

### **4.2 Conditions for Triggering a Fagan Inspection**

A Fagan Inspection can also be requested if problematic issues exist that informal reviews have failed to resolve. They can also be triggered as the project nears the point of a major release of the production code to look at key sections of the code.

### **4.3 Informal Review**

This review also serves as a means of spreading the knowledge to people other than

the primary developer of that section of code. This can be a less in-depth look at the software than a formal review, but if requested an in-depth look at specific parts will be done. In general these reviews will be populated by members of the Data Acquisition System software development team. Typically the code and release reviews will be of this type.

### **4.3.1 Informal Review Structure**

For an informal review, generally members of the development team will be asked to look over sections of the code or documentation they did not write and prepare comments or questions on it. They may be given a set of issues to consider ahead of time, but that is not required. Where possible at least two people should look over the same material, but there does not need to be complete overlap. Also this does not need to be an in-depth study of the code, but can be confined to a more casual review. The main point of the informal review is to insure nothing obvious has been overlooked.

There review period should have a defined length of time, two weeks is good as it allows time for people to work around schedule conflicts, and people should be told when the follow up meeting will take place. At the follow up meeting people should raise their concerns and these should be discussed. Difficult or contentious issues can be deferred to a later time to keep the meeting from lasting more than 2 hours. This type of review should be used to determine if code is ready for release as part of the normal development cycle and should be viewed independently of a formal review. In other words, a formal review does not eliminate the need for an informal review for declaring the code ready for release.

### **4.3.2 Informal Review Procedures**

The informal review should begin with the calling of a meeting of the development team once the development and testing has been finished, or when there is a specific issue a developer wants to discuss. At the initial meeting the team will discuss the focus of their efforts and whether it will be required for members to go off and study things on their own. If there is nothing contentious and everything seems in order, the process can be completed with a sign off during this initial meeting.

If independent study is needed, then the team will go off an review their assignments in preparation for the follow up meeting. At the follow up meeting the team will discuss what they learned and try to reach decisions on the issues. If not all issues can be resolved the cycle may have to be iterated. Also, if a problem is identified that needs addressing the cycle may be reverted back to the code release testing phase. If all issues are resolved this process can be

completed with a sign off during the meeting.

## **4.4 Fagan Inspection**

The Fagan inspection is a structured review with participation from outside of the development team. Whereas it would be beneficial to inspect all of the code at each release, that is likely not a practical scenario. Instead inspections will typically be done close to the final or near final release of the software. Because of the size of the code involved, these inspections will need to be confined to smaller key sections of the code or on issues which pose high risk for the system.

At least one member of the inspection board should be from outside of the project. This also limits the number of inspections that can be accomplished. Guidance on performing Fagan inspections can be found at the following locations:

<http://ods.fnal.gov/ods/www/process/ptr.html>

and

<http://ods.fnal.gov/ods/www/process/description.html>

### **4.4.1 Fagan Inspection Structure**

The Fagan inspection board will consist of typically 2 or 3 members, at least one of which is from outside the development team, the presenter, the inspection leader and the scribe. That is a total of 5 or 6 people for the inspection. Experience has shown that it is vital for everyone involved to be cooperative, and not to combine roles of the participants.

The presenter is the primary developer. They are charged with preparing packages of source code and making a presentation to the inspection team. They also are charged with answering questions of the reviewers openly and be cooperative. The presenter will also be expected to make a written response to each action items (e.g. suggested change in code), on what action they plan to take if any. This response should cover each item explicitly. Note the inspection team members only make suggestions, they cannot make mandates.

The leader runs the inspection and organizes the meeting. They are there to control the meeting and insure the meetings do not stall out or last too long. They must be assertive during meetings and make sure the process stays on schedule. They are also charged with reminding everyone involved at the start of each meeting that the aim of the inspection is to improve the software and no

one should take any comments personally. The software is what is being reviewed and not the developer. The leader should never be a line manager of the presenter, and results of the inspection should not be given to the presenter's line management (assuming we have enough people to exclude a line manager from the inspection and a line manager is not one of the developers).

The scribe is charged with taking notes during the meeting, preparing and sending those notes to the leader after the meeting for appropriate dissemination. Any comments by members of the inspection team on things to fix are to be noted by the scribe, not the leader and not the presenter.

The structure of the inspection consists of several parts: the presenter preparing for the initial meeting; the initial meeting where the presentation is made and inspectors receive materials and ask background related questions; the inspectors on their own examine the code; a final meeting where the inspectors present their findings and discuss them with the presenter; presenter writes a formal response to questions and suggestions and gives that to the leader of the inspection team.

#### **4.4.2 Fagan Inspection Procedures**

The first step is when a decision is reached indicating a Fagan inspection is desired. The initial interest for an inspection can come from the developers involved or the leader of the project team. The decision for an inspection can be made by the project leader or from higher up the management chain. The project manager can accept or reject a request for an inspection from the primary developer, but it is hard to imagine a set of circumstances when the project leader would want to reject this request for help from the developer. Before one can be sure an inspection is necessary, they should have a clear idea of what the review should focus on: specific sections of code; specific issues or concerns. Inspections should be confined to 1000 lines or less of actual code.

Once the project has approved the request for a Fagan inspection, the project leader needs to recruit the members of the inspection board. First step is to inform the presenter that an inspection is being organized and they need to prepare source code and a presentation. Discuss with the presenter what the focus of the inspection will be.

The next step is identifying the inspection board leader who will be charged with assembling the inspection team. The inspection leader needs to know what the focus of the inspection will be on so they can know what technical expertise would be most beneficial. Members from the development team who could be members of the inspection board should be given to the inspection leader once identified, and the name of the presenter should also be identified to the inspection leader.

The inspection leader will then try to assemble the Fagan inspection board. The

inspection leader may need to talk with the line management of people to find an outside member for the inspection. At this point one of the inspection board members will be chosen as the scribe, they do not inspect the code but instead take notes at the meeting on what has been said and write up a report that is given to the review leader. Once an inspection board has been assembled the inspection leader arranges for the initial and final meetings (a couple of weeks apart).

During the initial meeting the inspection leader runs the meeting. They are responsible for giving the charge to the inspection board members. The presenter will make a presentation and answer questions the inspectors have, while the scribe takes notes. The leader makes sure the meeting stays on track and does not extend beyond 2 hours. The presenter should indicate places in the code and issues they would like the reviews to give special attention. The leader adjourns the meeting at the end and reminds the inspectors they need to complete their study by the next meeting.

The inspectors now go off on their own and study the code, focusing on the issues and sections identified during the initial meeting. The scribe prepares a report for the leader based on their notes from the initial meeting.

At the final meeting, the inspectors ask questions and make comments based on what they learned. This is the first opportunity for inspectors to learn what their fellow inspection board members found. This meeting is a discussion among the inspectors and the presenter with the leader there to make sure the meeting stays on track. It is very important for the scribe to take careful notes of all suggestions and comments made that result in action items. The presenter is not there to defend a particular viewpoint or make a decision on how to respond to recommendations. Instead they are there to clarify issues and be part of an open discussion. The leader will adjourn the meeting and thank everyone for participating. This completes the involvement of the inspectors.

After the meeting the scribe prepares a report of all the action items and suggestions from the meeting. This report is sent to the leader of the inspection board. This completes the responsibilities of the scribe.

The leader will now discuss the report with the presenter. The presenter will then go off and prepare a response to every action item and recommendation in the report. How to respond to each recommendation is at the discretion of the presenter. They could say for each item in turn that they plan to not reject the recommendation. This information will only go back to the leader of the inspection and no further (definitely not back to line management). If the presenter refuses to be cooperative in the process, then they ultimately will gain no benefit from it. However the inspectors will have gained experience in how other developers analyze problems and perhaps new insights and knowledge because of it.

## **5 Software Languages and Environment**

The development of the Data Acquisition System will require the use of one or more software languages that will be used for writing the source code. The languages selected provide a framework for converting human readable instructions into a format the operating system can execute. The operating system and the user interface to it are referred to as the software environment in this document.

### **5.1 Software Languages**

There are a variety of software languages that could be used to develop components in for the Data Acquisition System. All of these languages have strengths and weaknesses and no one language seems ideally suited for all aspects of the system. There is no requirement that the system be developed using one software language throughout, and instead the language used for a given part of the system should be one best suited for the task given the available options. The current list of candidate languages are C/C++, Java, Python and R (for mathematical studies and plotting).

#### **5.1.1 Software Languages for Data Flow Applications**

Of the leading candidates for use in the Data Acquisition System, C/C++ provides the best performance for event buffering and transport mechanisms. Since this Data Acquisition System has fairly high rates between the Data Combiner Boards and the Buffer/Processor nodes, and there are requirements for fast buffering of hundreds of Megabytes in the Buffer/Processor nodes, C/C++ is the choice for these parts of the system. C++ code should be used as much as possible, with C code used only where necessary for performance reasons.

#### **5.1.2 Software Languages for Display and Control**

There is not a clear single language of choice for the display and control parts of the system. The choice of languages for the Display Systems, such as event display and monitoring, should be determined in part by what languages are compatible with the underlying display libraries. The underlying display mechanisms have not yet been determined and event display could use a different mechanism than event monitoring and DAQ monitoring. The likely choices are Java and C++ for the display applications and for the control applications like Run Control.

#### **5.1.3 Software Languages for Small Tasks**

There may be small tasks for various purposes that need to be accomplished in the Data Acquisition System. The choice of which language to use depends on the task. The choice between compiled code and scripts in this case should be based

on reliability, performance, ease of use, and ease of maintenance. Tasks that are to be performed by experts on an as needed basis are often well suited for scripts. Tasks that run all of the time, or periodically, often but not always are best accomplished with compiled code. Java, C/C++ and Python are the primary languages for these tasks. For a further list of scripting language options see the section of Software Languages for Scripts below.

### **5.1.4 Software Languages for Scripts**

Scripts in general should be written in Python. If for technical reasons Python is not a good choice for a particular task, then the script should be written in the Bourne Again Shell (bash). One exception to this rule involves mathematical analysis and plotting scripts. When a script is being developed for diagnostic purposes and requires mathematical manipulations, especially statistical methods, and plotting, then it is desirable to do the manipulations and plotting in R. The R language was developed for statistical analysis and plotting and provides a rich palette of tools to the developer for these types of tasks.

## **5.2 Software Environment**

The software environment represents the basic environment available to a user for interacting with the computer hardware. This consists of both the operating system and the command line or graphical user interface (GUI) available for interacting with the operating system.

### **5.2.1 Operating Systems**

There are two operating systems supported for the Data Acquisition System. The first is a standard Linux distribution repackaged by the Fermilab Computing Division to include enhanced security and features. This version will also include a custom built kernel to provide support for TRACE (a Fermilab developed message logging system for real time application debugging) and any performance tuning that is determined necessary. Since operating systems and kernels are evolving, it is not possible to specify the performance tuning changes to the kernel that are necessary at this time. This operating system will be used in both build and runtime systems.

The second operating system will also be based on Linux, but will be one tuned explicitly for an embedded processor environment. Although providing a reduced set of tool and potential libraries, this second environment is expected to be used for runtime purposes only and not as a build environment.

### **5.2.2 User Interfaces to the Operating System**

An effort will be made to minimize the number of user interfaces to the

operating system supported in the Data Acquisition System machines. Different machines in the system perform different functions and the interfaces supported are ones that map well to the functions required.

In the main systems which handle the data flow we want to reduce overhead where possible while still providing the needed functionality. These machines will not be running X servers as those servers provide overhead and latency while there is no need anticipated for graphical displays. The user interface thus will be a command line shell environment on these nodes. For consistency sake the supported shell environments will be the Bourne Shell (sh) and the Bourne Again Shell (bash), with bash the preferred shell when possible. These two shells use very similar in syntax with bash extending and enhancing the sh functionality.

## 6 Documentation Standards

Do we have a preferred text editing tool?

Are documents going in CVS and/or documentation database?

## 7 Naming conventions

This section covers various naming conventions for product structure. To distinguish between compiler versions used, the products will have a compiler version encoded as a UPS qualifier when the product is installed.

### 7.1 *Directory structure*

Product instances shall follow the following directory structure where applicable:

- bin/  
All non-platform specific files that are needed to run the product.
- bin/ppc/  
PPC specific binaries that are needed to run the product.
- bin/x86/  
X86 specific binaries that are needed to run the product.
- config/

This directory is where configuration files used in building or running the product reside. This includes a configuration file for doxygen. PyLint configuration file resides in this directory. There should at least be a

configuration file for JCSC (Java code source checker).

- doc/  
Where documentation goes. This is where both the user guides produced by hand and the automatically generated documents from doxygen will go. Doxygen will create subdirectories for latex and html versions. Note that running Doxygen (1.4.6 or later at least) produces a make file in the latex area that can be used for generating a pdf version.
- lib/  
All non-platform specific libraries that are needed to run the product (e.g. Java Jar files).
- lib/ppc/  
PPC specific libraries that are needed to run the product.
- lib/x86/  
X86 specific libraries that are needed to run the product.
- test/  
Anything related to testing the product goes here. This test directory will have the same subdirectory structure as the product itself – eg, bin, config, cxx, python, etc directories.
- tools/  
Any executables or scripts that are needed for building (not running) the product.
- ups/  
UPS table file goes here.

### 7.1.1 C/C++

If multiple namespaces are used in a product, the directories listed below may have substructure based on the namespace. However, this choice is left to the product developer(s).

- cxx/dep/  
Any dependency files created during the build process.
- cxx/inc/<product>  
The include files can reside one level down from the inc directory in a subdirectory with the same name as the product. When setting the include path, we would use the path to the "inc" level and then include files using

<product>/<include\_file>.

- cxx/obj/ppc/  
PPC specific object files.
- cxx/obj/x86/  
X86 specific object files.
- cxx/src/  
Source code for libraries and binaries that are used to run the product.

### 7.1.2 Python

- python/src/<product>/  
The main files for the product go here.

### 7.1.3 Java

The naming conventions for Java packages should be as follows:

- package gov.fnal.nova.<product> for nova-specific code
- package gov.fnal.cd.<product> for experiment-independent code

With these conventions, the package paths (<package-path>) will be gov/fnal/nova/<product> and gov/fnal/cd/<product>. The directories listed below include the package path when appropriate.

- java/classes/<package-path>/  
This is where javac would place the .class files it builds. We may only want this for some testing and for the rest of the time rely on building and using jar files.
- java/src/<package-path>/  
This is where the .java source files reside.

## 7.2 ***File Naming Convention***

General file naming conventions should follow the rule of the filename matching the class name defined in the file and preserve case sensitivity. Apart from nested classes, only one class should be defined in a file.

Files for tests, and thus the classed defined inside, should append the string “Test” to clearly distinguish them from other classes.

**Expand on this.**

## 8 Software Tools

This section covers tools selected for use in the development of the Data Acquisition Software for the NOVA experiment. The software tools are designed to aid in the development and debugging and quality assurance of the systems. These tools include compilers, debuggers, code validation, build systems, code management, automated documentation generators, profiling and integrated development environments.

### 8.1 Compilers

Source code can be interpreted, compiled bytecode which is interpreted, or compiled to binary code. Python and bash shell scripts are interpreted and do not need to be compiled in the usual sense.

#### 8.1.1 C/C++ Compiler

The C/C++ compiler will be gcc/g++. There are version issues that will need to be taken into account when selecting a compiler version due to changes in name mangling between some versions for C++ code. There are third party tools we will be using that are thus tied to a particular compiler and version combination and we will need to make sure compatible versions of the third party tools are available before migrating to a new compiler version.

#### 8.1.2 Java Compiler

The java compiler comes with the Java development toolkit. The version should be at least 1.5 and we should aim for starting with the newest stable version available at the time Java code development starts. The Javadoc application also comes as part of the toolkit so version compatibility is not an issue. There may be issues however with third party tools that will be used, and so this will need to be evaluated before any switch in versions is officially made.

#### 8.1.3 Python Compilers

Python scripts are compiled into bytecode (I think, but not the same bytecode used by Java) on the fly when imported by other scripts, and this compiled data can be cached in files for future imports, but generally are not compiled to the machine code level. Essentially there is no separate phase of a project for compiling Python code for use, although one can create all the bytecode files ahead of use by importing all of the scripts before distribution.

There is also a possibility of compiling Python scripts to improve performance, however for this project cases where that becomes necessary should trigger a re-write of the code in Java or C/C++. There is also the possibility of creating Python modules from C code, in those cases the C/C++ compiler should be

used.

## **8.2 GUI Tools**

asdf

## **8.3 Authentication/security**

During the development cycle of the data acquisition components it is important to identify potential places where network security can be applied. Data encryption and peer authentication may be required to strengthen network security.

There are some options for encryption and authentication, many of them are available in the Grid Security Infrastructure (GSI), which is part of the Globus Toolkit (<http://www.globus.org>).

Authentication of users and services is handled through certificates. Certificates contain information that is used to identify users and services, and are issued and verified by a trusted Central Authority (CA).

Data encryption is used during the authentication phase, but it is not clear whether Globus provides mechanisms for encrypting socket messages (e.g. run control message to buffer nodes). For reliable and secure file transfers the GridFTP service can be used. The GridFTP user authentication also uses certificates.

The Globus Toolkit provides interfaces for C/C++ and Java only, but there are interfaces in python available from other packages.

## **8.4 Debuggers**

### **8.4.1 C/C++ Debuggers**

There is a C/C++ debugger called gdb that comes with the gcc/g++ compiler. This is a command line tool and does not provide any GUI support. There is a GUI debugger that called ddd that can use gdb as a backend and this is can be used for some debugging situations. However it is not very easy to debug in a threaded environment using gdb, and so when possible it is desirable to use the TotalView (<http://www.etnus.com/>) debugger. Because of name mangling issues there can be C++ compiler incompatibilities encountered.

## 8.4.2 Java Debuggers

Java code can be debugged by importing the software into the Eclipse IDE and running the debugger it provides.

## 8.4.3 Python Debuggers

Currently one way to debug Python code is to use the pdb module. This allows one to set break points in the code and then run the script and step from there. It is not a great solution but it can be helpful when using print statements, the typical Python method of quick debugging, isn't sufficient. This is recognized as not ideal. If there is significant Python development on the project, then a search for better debugging tools for Python would be wise.

## 8.5 *Build Systems*

### 8.5.1 Build System for C/C++

The primary build system for C/C++ will be make. Each package is required to provide a Makefile which minimally defines targets for: building object files; building libraries; building executables; cleanup to remove built objects.

If a more advanced and easy to use build system is evaluated and found sufficient, then it will be considered as a replacement or parallel build mechanism.

### 8.5.2 Build System for Java

The primary build system for Java will be make. Each package is required to provide a Makefile which minimally defines targets for: building class files; building jar files; building executables; cleanup to remove built objects.

If a more advanced and easy to use build system is evaluated and found sufficient, then it will be considered as a replacement or parallel build mechanism.

### 8.5.3 Standard make targets

The following standard make targets should be supported by each make file. Additional targets may be used in products that have non-standard build steps such as automatically generating source files. The selection of the target C++ architecture is handled with the TARGET\_PLATFORM environmental variable which should be set before running make.

- all – this target builds the full product including Java Jar files (if needed) and C++ libraries and binaries for PPC or X86.
- bin – builds all of the binary files that are needed to run the application.
- bincxx – builds the C++ binary files needed to run the application on PPC or X86 platforms.
- binjava – builds the Java binary files (unlikely).
- clean – removes as many of the derived files as needed to ensure a clean build when one of the other targets is next built.
- cleancxx – removes the derived C++ files.
- cleanjava – removes the derived Java files.
- docs – builds the Doxygen documentation (and any other derived documents).
- lib – compiles all library source code and stores the results in the appropriate library archives (equivalent to “libcxx libjava”).
- libcxx – compiles the C++ code for PPC or X86 and stores the object files in one or more library archives.
- libjava – compiles the Java source files and stores the class files in one or more Jar files.
- test – generates all of the unit and system test applications for the product (equivalent to “testcxx testjava”).
- testcxx – generates the C++ unit and system test applications for the product.
- testjava – generates the Java unit and system test applications for the product.

## **8.6      *Integrated Development Environments***

There is no requirement to use an Integrated Development Environment (IDE) for this project. However if a developer decides to use an IDE, then the code must be easily imported into and out of that environment. The IDE must also in no way impose restrictions that would cause the source code files to be incompatible with any of the following: Code Management System; Build System; Automated Documentation Generators; Source Code Checkers or Coding Standards; Compilers; Debuggers. One possible IDE that appears to meet these requirements is the Eclipse IDE. An additional requirement is that any developer who does use an IDE must document how to: import and export packages from the environment; install IDE and any plugins if necessary; how to use IDE for developing software

for the environment.

## **8.7 Automated Documentation Generators**

Automated documentation generator parse and pre-process source code files to extract documentation tags and build reference manuals for the code. This level of documentation provides the most benefit to the developers and maintainers of the software, and is invaluable when it comes to handing off support to someone who was not on the project during the development phase. The advanced generators have built in pre-processors that understand the language syntax. These generators also understand a set of tags which are used to document the functionality of classes, methods and functions, etc. The tags are very useful in defining the purpose of methods as well as the meaning of arguments and exceptions thrown. Where supported by the generators, Javadoc style tags are required in the code. Since the documentation is for the developers and maintainers, it is sufficient to produce web based documentation (html files).

### **8.7.1 Automated Documentation Generator for Java**

Instead of using the javadoc application which comes with the Java installation for documentation generation for Java code, Doxygen, as of version 1.4.6 at least, is believed sufficiently mature for this purpose. If in the future Doxygen is found to not properly handle the formatting for this project, then Javadoc will be used as an alternative. The use of Doxygen for Java will allow generated documents to be consistent with those generate for C++.

### **8.7.2 Automated Documentation Generator for C/C++**

The Doxygen program should be used to generate documentation from C/C++ source files. Doxygen uses a configuration file to determine most of the behavior on what is documented and what is ignored. There should be a standard template for C/C++ packages produced and each package should have a tailored version resident with the code. As much as possible, settings should be used that provide formatting as close to that of Javadoc as possible. There are multiple switches available that deal with making the behavior similar to Javadoc.

### **8.7.3 Automated Documentation Generator for Python**

Doxygen 1.4.6 can parse Python code and produce documentation. here are two ways to add documentation now for Python code. The first is to use the triple quote method. Blocks enclosed in triple quotes will be processed by the python

interpreter via the doc() method call. This allows users to get interactive help on a class or method and is very useful during the development phase. Doxygen will also parse these blocks as a whole chunks and put them in the documentation it produces. However, it does not recognize tags inside the triple quotes so you don't get nicely formatted documentation but instead get a block of text pretty much as you entered it.

There is a second method recognized by Doxygen which will parse the special tags (e.g. @param). This leads to much nicer documentation. The basic structure is to start a comment block section with "## " and then subsequent lines in the block start with "# ". Also, the comment blocks need to preserve indentation of the executable code otherwise they are ignored. Below is an example that worked, but not necessarily with the text we would use (Class Definition, etc is based on a different way to parse the file).

We want to preserve the utility of the doc() method calls (triple quote documentation scheme), but also want to have the more useful structure, which is also more consistent with the C/C++ format, for the documentation. Therefore we may have to put in a bit of duplicate information within the python source files to achieve this. I haven't been able to think of a good way of doing this without the duplication.

Below is an example of how one could handle python source code documentation. Note the Doxygen would like to see the self variable explained with the @param tag, but that would not be in line with method signatures for C++ or Java.

```
## Class Definition: partition Class ---
#   Class for tracking state of a partition file in dlsam.
#
class partition:
    """
    This is a class for tracking the state of a partition file and
    the description comes from the __doc__ string.
    """
    ## Class Method Definition: partition.__init__ ---
    #   Initialization of class instance for partition Class.
    #   @param info Imported state dictionary object
    #   @returns None
    #
    def __init__(self,info):
        """
        Constructor for the partition file state tracking class.
        Description
        from the __doc__ string.
        Inputs:
            info Imported state dictionary object
        """
        ## @var historyDict
```

```
# a history dictionary
self.historyDict=None
```

### 8.7.4 Languages without Automated Documentation Generation

There are no documentation generators we are aware of at this time for scripts written in bash or R. These scripts should still use Javadoc style tags to document the code. Minimally a parsing program could be provided to pull out these tags and generate some simple documentation based on the comments alone. While this would not have the benefit of pre-processing and understanding the underlying script syntax for structuring, it would still provide some useful documentation. Regardless of when the generator is made available, these scripts will still be required to properly use the tags.

## 8.8 *Profilers*

There is need for a profiling tool that runs under Linux and handles threaded applications properly.

### 8.8.1 C/C++ Profilers

An evaluation of oprofile was performed in 2005 and it was raised strong concerns about the accuracy matching time to the proper thread and method. A tool has been developed at Fermilab to do a better job on time accounting for threaded applications and this tool is considered the best choice at this time. This tool is called perf\_tools (<http://home.fnal.gov/~jbk/profiling/index.htm>).

### 8.8.2 Java Profilers

There have been no specific Java profiling tools identified at this time. The Eclipse IDE continues to be enhanced and may provide this functionality by the time development starts. It should be noted C/C++ will be used for high performance situations which might benefit from profiling tools, so this may not be an issue for the Java code for this project.

### 8.8.3 Python Profilers

There may be Python profilers available, and if so these could be used to identify parts of the Python code that should be implemented in C/C++ or Java. However Python applications that demonstrate performance issues should in

most cases just be implemented in C/C++ or Java and so profiling is less of an issue.

## **8.9 Source Code Checkers**

There is varying support for checking standards like naming conventions and other programming practices. Thus some of this will need to be done by hand. However, were available tools should be used to aid in the process.

### **8.9.1 C/C++ Code Checkers**

For C/C++ programs there does not appear to be a Linux version of a program like the old Unix lint program. Thus some of the naming convention checking may have to be done by hand. There is an application that does a good job of checking for programming errors such as memory leaks. The application of choice for this task is Insure++ (<http://www.parasoft.com/jsp/products/home.jsp?product=Insure>). Because of name mangling issues, the version of this application can be depended on the compiler version. Naming convention standards may need to be checked and enforced by hand.

### **8.9.2 Java Code Checkers**

There is an application available for checking coding standards in Java called JCSC (<http://sourceforge.net/projects/jcsc>) which should be used for validating all Java code developed for the Data Acquisition System.

### **8.9.3 Python Code Checkers**

There is an application, modeled on the old Unix lint application, for checking coding standards in Python scripts called PyLint (<http://www.logilab.org/projects/pylint>). This application should be used for checking Python scripts for the Data Acquisition System.

## **8.10 Code and Build Requirements for Aiding Debugging Process**

This section deals with requirements designed to aid in debugging the system. In order to use debugging tools, the software needs to be built with enough information included for the debugging tools to properly map the instructions back to the source code. Thus it is required that all libraries be built in debug mode and not in any optimized mode.

To help with diagnostics before resorting to a debugging tool, use of the TRACE

package should be used in key sections of the code for the primary C/C++ applications to insert messages into a circular memory buffer. The Linux operating systems on all machines in the Data Acquisition System must run a kernel patched for TRACE support.

### **8.10.1 Trace Levels**

Trace levels shall be defined as:

## **8.11 Code Distribution**

Our experience with developing, commissioning and long term operating and maintaining of Data Acquisition Systems has highlighted the importance of multiple concurrent version support. The ability to deploy a new version into a running production environment without disturbing the operations of the system is highly valuable. This allows the support team to deploy a version ahead of time, configure it for production use, and schedule a switch over for a later time. It also allows for a rapid rollback to the older version, without needing to install or uninstall any software, if an unforeseen problem should arise in the production environment. It also allows for simultaneous debugging on one machine of multiple versions of an application which can be invaluable when investigating subtle problems.

### **8.11.1 Issues with RPM for Multiple Version Support**

Since we expect the operating system to be based on a Fermilab packaged version of Linux, and that version is an rpm based system (RedHat Package Management), a logical system to consider for code distribution is the rpm mechanism. However, natively the rpm mechanism does not support simultaneously installed versions of the same package. This system is designed for management of systems to a consistent level of software packages, but not for multiple instances of the same package. The idea is to allow libraries and executables to be installed in centralized locations so the user environment does not have to point to special areas to pick up the libraries and executables. While that allows for easy definition of a user environment and maintenance for one version of a package, that model does not at all work for allowing easy access to multiple versions of the same package.

There have been proposed ways to work around this limitation by including version information explicitly as part of the package name, using multiple rpm databases and installation target areas, and similar kludges. However, even ignoring the many issues of maintenance these types of schemes would raise there is still the issue of being able to quickly define the user environment to point at the correct executable and library versions. There would need to be a

mechanism for quickly and easily configuring a user's environment for a different version of a software package. This system would need to allow the same user to be configured to use different versions in different shell windows at the same time in order to support simultaneous debugging. The system would also need to provide a quick and easy to understand means of determining which versions of the package are available on the system and an easy way to select the desired one. Finally there would need to be easy to use infrastructure to maintain these multiple versions, install new ones and uninstall old ones all without placing a large burden on the developers or support staff. None of this functionality currently exists and it all would have to be designed, developed and maintained.

For these reasons the rpm mechanism is not well suited for the short, medium, or long term needs of this project.

### **8.11.2 UPS/UPD System for Code Distribution**

The UPS/UPD, Unified Product Setup(?) and Unified Product Distribution(?), was developed at Fermilab to provide the easy to use and maintain multiple package version support. The system is no longer in the development phase and is considered supported only through the end of Run II at this time. However we do possess a great deal of expertise and experience with this system and it would exceed all of our requirements for this project.

The long term support is an issue, however we do have on our team one of the original developers and they believe they can continue to internally support new builds as needed for this project. Therefore we believe the risk of using this mechanism is minimized by our access to the source code and ability to in house support the product.

UPS/UPD can be separated into two distinct sets of functionality. UPS provides support of multiple versions of packages on a system along with functionality easily determining versions available, installing or uninstalling versions, querying for available versions, and configuring the user environment on the fly to switch between versions. Complex dependencies between packages and actions for configuring the user environment are supported, however we envision only using the minimal (and thus easily support) features.

UPD is a product distribution mechanism that uses a centralized ftp repository for storing package versions for distribution to multiple hosts. The system supports easy access to lists of versions of packages available as well as seamlessly integrating with the UPS mechanism for installation at the remote sites. While this is a valuable mechanism for code distribution management and we would certainly use it if we can, we can also easily live without it. We already know how to independently package software for the UPS environment, and integrate with the UPS mechanism for installing the software independent of

UPD. This has been previously accomplished with scripts in another project where some packages were maintained outside of UPD. Therefore there is little risk in selecting UPD as a possible code distribution management system.

## **8.12 Code Management Systems**

Software development for this project will be a joint effort. Thus there may be multiple people working on the same application or library at any given time. In order to manage this environment and reduce the risk of changes being lost, a code management system will be needed. This system should provide a central repository that can be accessed from different machines by different developers. It should also provide a means for merging in changes as well as tracking older revisions for future reference and the ability to group versions of various files with a name so that a complete set of files as a snapshot in time can easily be accessed.

In the Fermilab Computing Division there is a lot of experience with CVS for providing this type functionality. CVS fulfills all of the requirements for this project. There is also centralized support for a CVS repository at Fermilab, so by using that system we eliminate the need for maintaining the repository and infrastructure ourselves. Given all of these benefits it is clear that using CVS as the Data Acquisition code management system is a good decision.