Senior Project Report - DocTest

Stephen Weessies

Computer Engineering

California Polytechnic State University, San Luis Obispo

March 24, 2013

Abstract

DocTest is a program that, simply put, allows a programmer or user to document STANAG 4586 (a standard for unmanned aerial vehicle interoperability) messages and test the vehicle system at Lockheed Martin [5]. The program is extensible to allow for further development aiding our software team to do what they do best and not get bogged down in tedious but necessary documentation. DocTest is also used to aid in testing, keeping track of the issues and bugs found and creating a document that captures each issue so an issue is not missed or forgotten. This program was made for use at Lockheed Martin, and as such some aspects of the program cannot be discussed in this document.

Contents

List of Figures

Figure 1: Table of acronyms and definitions that are used throughout this paper

Deserialized	The process that extracts data from a file and creates an object.
DTP	Detailed Test Procedure
HITL	Hardware in the Loop
IDD	Interface Description Document
IP	Internet Protocol
JSON	JavaScript Object Notation
NATO	North Atlantic Treaty Organization
Serialized	The process that takes an object and parses it to a file format.
SIL	Software Integration Lab
STANAG 4586	Standard Interface for UAV interoperability.
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol, a type of IP packet
Winforms	Windows Forms (older style .Net controls)
WPF	Windows Presentation Foundation

1 Intro: The Need

I have been working at Lockheed Martin UIS for over 3 years; a group that was previously called AME UAS Inc. While the software team is small, large programs are worked on constantly that require maintaining very strict quality control to make sure no problem is left un-fixed. This process tends to be prone to error sometimes as it is difficult to keep track of each software package used in the aircraft or ground-station. The documentation aspect of a programmer's job tends to be the one thing every programmer dislikes. However, this documentation is very important and is necessary in many government programs that are taken on. Having good documentation can be the difference between getting a contract extension and the contract ending. Documentation is important and the style of each developer differs. I proposed to make a program to aide in keeping Lockheed Martin's documentation always up-to-date and correct. This program will keep the formatting and style the same across different documentation documents. This program will also allow for better documentation through linking several key aspects of the development pipeline together.

Our IDD (Interface Description Document) keeps track of all the STANAG 4586 messages that our software can handle. STANAG 4586 is a NATO standard that allows UAVs to use the same message scheme so that the systems can operate together. This allows for one UAV to be controlled by any other system that uses STANAG 4586. These messages are UDP based packets that have a set header. There are many defined messages that all UAVs should use to maintain STANAG 4586 compliance. The IDD is typically hand edited and is prone to having old message definitions or messages that are outdated. This is unexceptable as not only is the documentation wrong, but it can easily cause another programmer to create code pertaining to an old or deprecated format. Linking the code to the documentation was the highest goal of this program as many hours have been spent on fixing code that was designed to old documentation, as others forgot to update the documentation or made a mistake. By having a file that can store each message that we use we will be able to allow each program to use this specification and create the necessary code. DocTest, the program described in this report, will then be able to take these specification files and create documentation based on them.

The testing procedures that we implement at Lockheed Martin are called DTPs (Detailed Test Procedures). These tests tend to take several days and encompass that entire system both in a SIL (Systems Integration Lab) and in HITL (Hardware in the Loop). This usually allows us to find any bugs or issues before the aircraft is released for flight. Currently the testing procedures are printed out and placed in a binder. This allows for many testers to use the document while each test is being completed. However, keeping track of the issues is difficult as there are many pages of issue reports that contain both minor issues, like wrong color on a button, and all the way up to the aircraft crashes on launch. There is not an

immediate way to determine the severity of an issue, nor does the tester have a defined way to show what they think the issue's severity is or what the problem may even be. DocTest will allow each tester to keep track of the issues in the application and even have the ability to write a description on how the issues are exactly created, or what a possible solution or workaround may be.

2 Proposal Design Review Results

Lockheed Martin wants the quality of software to be the highest possible especially in our industry. As such, they are very proactive in implementing new procedures to insure better quality in software. DocTest was immediately accepted as a program to work on. I was given a large amount of time initially and was given a few other software engineers as resources to help test and critique the program. The original proposal gave an estimate of 95 hours of total time, to be used by all programmers, 70 of which were to be my hours alone. Shortly after this proposal was approved, we received a stop order on other work. This then allowed more free-time and this program's scope was greatly increased. As of this publishing date more features are being added to further enhance quality software at Lockheed Martin UIS.

3 Requirements

As stated previously, requirements were changed throughout the process as more time was allotted. The differences will be clear in the following two sections.

3.1 Original Requirements

1. The program shall store and manage the IDD and the messages used in the IDD.
   (a) The IDD shall be stored in HTML for easy linking.
   (b) The IDD shall allow descriptions and images to be added for each message.
2. The program shall revision control the IDD.
3. The program shall send and receive STANAG 4586 messages.
   (a) Further, the program shall show statistics of traffic on the incoming and outgoing connections.
4. The program shall allow Python scripts to be run from within the application.
5. The program shall manage the DTPs and create a digital DTP to be used with testing.

6. The program shall allow users to do the DTPs in the program.
   (a) At the end of testing, the program shall save the DTPs in an HTML file for easy access.
   (b) The program shall link each issue to the corresponding test procedure.
7. Each issue found shall have a severity level associated with it, and must be one of the following types: Note, Bug, and/or DTP issue.
8. The program shall be written in C# within Visual Studio 2012 and must run on Windows 7+ machines.

3.2 Final Requirements

1. The program shall store and manage the IDD and the messages used in the IDD.
   (a) The IDD shall be stored in a PDF[1] created using LaTeX[4].
   (b) The IDD shall allow descriptions and images to be added for each message.
   (c) The IDD shall allow descriptions and images to be added for each message group.
2. The program shall revision control the IDD.
3. The program shall manage the DTPs and create a digital DTP to be used with testing.
4. The program shall allow users to do the DTPs in the program.
   (a) At the end of testing, the program shall save the DTPs in an HTML file for easy access.
   (b) The program shall link each issue to the corresponding test procedure.
5. Each issue found shall have a severity level associated with it, and must be one of the following types: Note, Bug, and/or DTP issue.
6. Each test shall record the person(s) that is/are testing.
7. The program shall be extensible to add more features easily in the future.
8. The program shall be written in C# within Visual Studio 2012 and must run on Windows 7+ machines.

3.3 Major Changes

Above are the two sets of requirements. The first being the proposed requirements and the second being the set that were generated after getting more time allotted to the program to further enhance the quality of the software and it's outputs. The major difference that took the most time to implement was a way to create PDF[1] documents using a language called LaTeX[4]. LaTeX is able to produce very high quality reports that stand out by having many features that would have been difficult to have in an HTML driven program. Removing the requirements for having DocTest handle Ethernet traffic did save some time; it was deemed that this feature would not add much to its intended audience. We already have tools that aid in this and it would have not been time well spent on implementing the same feature twice. The other large addition was the ability for DocTest to be extensible. This will allow for more development down the road and will allow for easier changes.

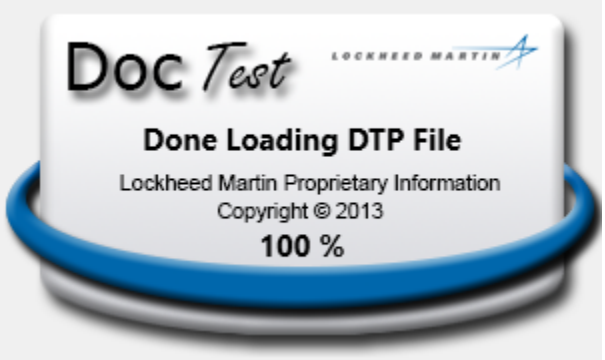
3.4 Why LaTeX

Originally HTML was planned as the output for DocTest, however a more professional document generation system was requested by management at Lockheed Martin. We have had some experience with LaTeX and decided that it would be the best choice for DocTest. The necessary amount of time to learn LaTeX was deemed to be acceptable as our documentation would be better accepted by the executive team and would show the professional level we desired. The formatting capability of LaTeX is what makes it the best choice. There are many ways to get exactly what is needed to be shown or laid out. When compared to Microsoft Word©[7], the layout options and formatting exactness are amazingly better in LaTeX. This decision to move to LaTeX has rippled throughout the whole company now as all of our documentation programs are now using it to keep the formatting and style similar and looking professional.

4 Functionality

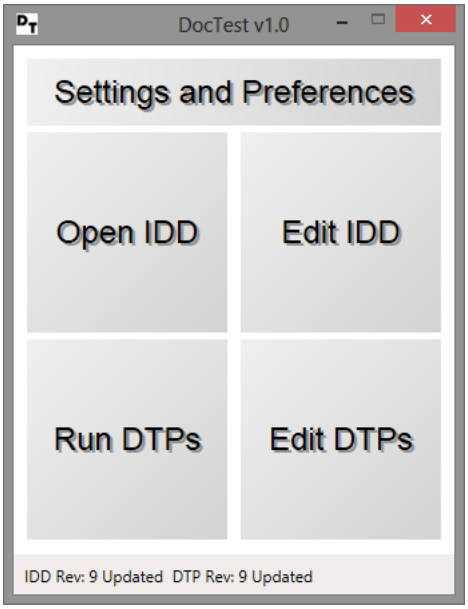
DocTest has two major portions currently, the IDD and the DTPs. The first window seen is the loading window. This will show the progress of updating and loading of both the IDD Specification files and the DTP Specification files, as seen in Figure 2. This process usually takes around 3 seconds but depends on the intranet or internet connection speed and some CPU power for loading all the data. The loading screen is custom designed to update whenever it gets a call. Most loading screens update at a given rate and thus can sometimes show incorrect loading data. If the program were to freeze the loading screen would show where the program froze and the details of where the problem occurred. This has happened several times and is another useful tool that can help programmers find bugs, while also maintaining a better looking program.

Figure 2: Loading screen for DocTest. This shows the progress of updating and loading the files for viewing or editing.



The Main screen then is shown once everything is loaded, as seen in Figure 3. This allows for opening or editing the IDD and running or editing the DTPs. Changing the location of the DTP and IDD files can be done in the settings window.

Figure 3: Main screen of DocTest. Everything starts here.



4.1 User Interface Design

Lockheed Martin has been making software utilities that have been very useful in allowing more freedom for users. Most of the applications have not allowed for many windows to be opened at the same time. This does not allow the user easy use of all data available. I decided that this needed to be changed. DocTest is able to have any of its windows open all at once. This allows the user to see all the relevant data from different sources and even allows the ability to copy and paste from one window to the other. The window designs also follow the design approach for the files. The DTPs and the IDD both have 3 levels of objects and similarly each have 3 levels of windows. This approach is called drill-down and can be seen in Figure 4. Each parent window has the ability to close its children. This can be useful when the user is done editing and wants to close all the windows at once. This is usually the issue with a drill-down approach but is easily mitigated through window relationships. This design was controversial when I first designed it. However, after implementation everyone on the software team loved it and we are already designing other applications similarly. The one downside to the design is that it is designed with programmers in mind, this could alienate other users. However, as this application is designed to be used only by the software team, this should not be a problem.

Figure 4: Multiple windows are open, showing how DocTest has a drill-down approach to the user interface.

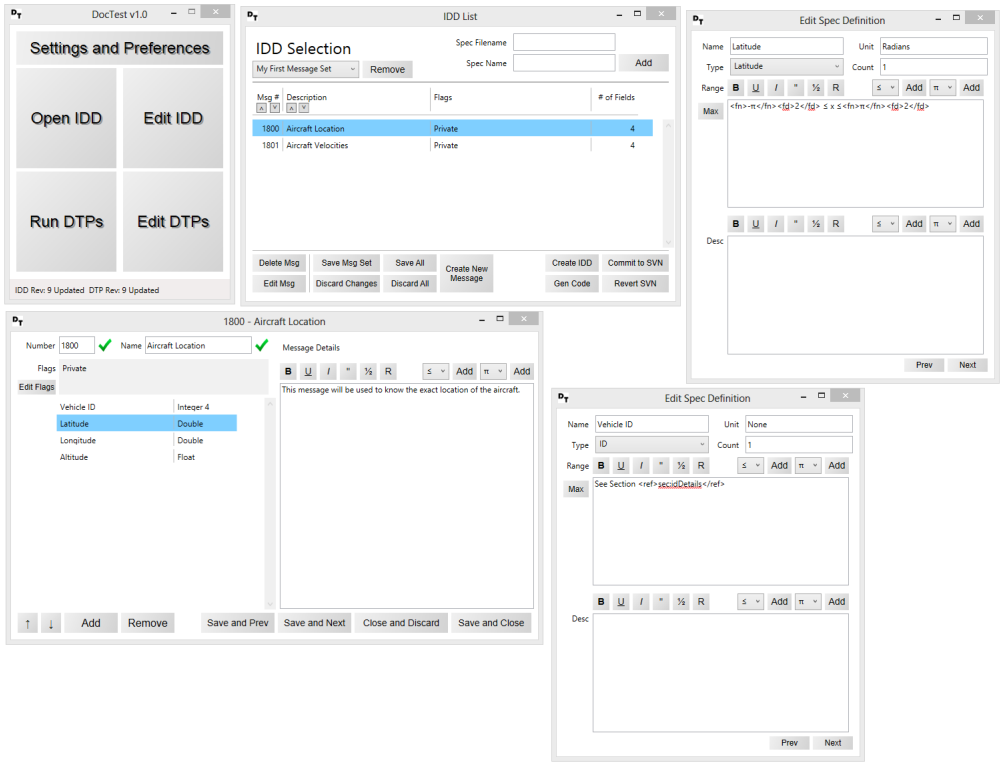


Figure 5: Main image for loading screen.

Another aspect that I wanted to make sure I improved upon from previous applications was the Loading Screen. It is the first thing that users will see and it is always good to put your best foot forward. Even though this application was only going to be used internally, I wanted to show everyone that used this application what we could do to improve our other applications. I have a background in Photoshop so I created an image that had curves and even had the progress bar curved at the bottom. A simple progress bar would have sufficed but having the nice bend in it made it more visually appealing. The progress is shown by overlaying the progress image over top of the main image in a sweeping motion from left to right, the two images can be seen in Figure 5 and Figure 6. The slight overlap in the progress is for the shadow to correctly show over the main image. Also, like most of us have dealt with, most progress bars are not balanced in their reporting of the actual progress. I spent some time trying to even out the progress bar to show it's best possible progress at any given time. Even though this process takes around 2 seconds it can be easily used in other applications that take upwards of 1 minute to load.

One final aspect that needed to be controlled was the similarity between all the controls. Different styles of the controls would both make the application look confusing and would also make it harder to develop or change anything in the future. WPF controls were to be used in DocTest. WPF is the Windows Presentation Foundation, this is a set of controls that are designed for C# and utilizes Microsoft DirectX©[6]. Most WPF controls have enough functionality already so they were readily used. However, Buttons did not have all the functionality that was desired. There is a possibility that Lockheed Martin will be buying new touchscreen devices. DocTest could benefit from touchscreen friendly buttons. There were a few areas where the button needed a toggle ability to turn on or off a function. These buttons handle both toggle and action button events, this can be easily set in the creation of each button. DocTest was to be a standard for other future Lockheed Martin applications,

Figure 6: Progress overlay for the loading screen.

Figure 7: Custom button shown when mouse is hovering over, with help text shown.

and as such DocTest should have buttons that look nice and have some effect to show when the mouse was over (Figure 7) or clicking on it (Figure 8). Transition effects were added to smooth this effect to make it more pleasant to the user, and add a more professional look. By creating this custom button DocTest was able to ensure that each button is uniform looking. Also, if something needed to be fixed in the custom button it would be fixed without having to fix the individual instance. Once this button was created many of the developers like the design. However, one of the developers was a little unsure what each button did. This was due to lack of training and not knowing enough of the system to understand the functions that were provided to him. Tool tips were then added, which when hovering the mouse over the button causes a small text info box to appear telling the user what the button will do. This is done programmatically and uses a smart filter to create the help text. For instance, if the button has the text, “Open IDD”, then the help text will show ”Opens the IDD PDF”. Keeping the same verbiage is a necessity with this addition, but it aides in not having to maintain multiple lists of help text and button text.

Figure 8: Custom button when the mouse is clicking on it.

4.2 IDD

The IDD is used to keep track of all STANAG messages that are in use in our applications. STANAG contains around 200 messages that are predefined, and we can then define more messages that are called private messages to add functionality. Our IDD currently contains around 400 messages, not all of which are in use. With the old IDD that was edited by hand, some messages that were deprecated were not marked as such and were believed to still be in use. This is a problem as it allows us to keep old code that may be hindering development. DocTest allows for deprecated messages and will keep them separated in their own section of the IDD to make sure they are not actively being used; they should only be used as reference.

4.2.1 File Structure

The IDD is stored in specification files using JSON. JSON is a human readable storage file, this allows for easy parsing and editing of files. JSON is widely used in many file storage solutions when easy editing and viewing is needed. JSON allows for easy cross-platform storage that can easily be created as an object in DocTest.

1. Name
2. Description
3. SpecItems
   (a) Message Number
   (b) Name
   (c) Description

(d) Definitions
    i. Type
    ii. Units
    iii. Range
    iv. Length
(e) Flags (what the message connects with)
(f) Resources (images used in the description)

The heart of the specification file is the definitions for each message. This is where each field is stored for each message. The type corresponds to a known type like doubles or integers. The units are used to show what format these should be shown in; sometimes SI units cannot be sent over a message as they can take up more bandwidth. When that is the case the unit will show what unit is being used for the data field. Having a different unit can be tracked so the program will know how to package the data correctly. The range field is similar and will allow for known ranges within the data type. This range can be used to allow for packing the data into a smaller data type. Temperature is a great example of a range that is typically limited. Temperature would usually be sent down as a double, however not much precision is usually needed and the field can be packed into a byte or short depending on the desired level of precision. If this definition has the need for a length, then the length field is filled in with the length of the field. This is especially useful when dealing with strings as they need to be stored as character arrays. The format is simple to parse using JSON and adding additional fields does not affect previous versions of files. This was useful when DocTest was actively changing and being used at the same time. In Figure 9 there is an example of what the JSON file looks like. Some formatting was changed to keep the code snippet to one page.

Figure 9: JSON file showing an example of what the IDD Specification file looks like.

```
{ "MessageSet": "My First Message Set",
  "Set": [ {
      "number": 1800,
      "desc": "Aircraft Location",
      "definition": [ {
          "Vehicle ID": [
            "id",
            1,
            "None",
            "See Section <ref>sec:idDetails</ref>",
            "" ]
        },{
          "Latitude": [
            "latitude",
            1,
            "Radians",
            "<fn>-$pi$</fn><fd>2</fd> $le$ x
            $le$ <fn>$pi$</fn><fd>2</fd>",
            "" ]
        },{
          "Longitude": [
            "longitude",
            1,
            "Radians",
            "<fn>-$pi$</fn><fd>2</fd> $le$ x
            $le$ <fn>$pi$</fn><fd>2</fd>",
            "" ]
        },
      ],
      "details": "This message will give us the
      location of the aircraft.",
      "flags": null,
      "resources": [] } ] }
```

Figure 10: Code showing the dictionary used to translate between Python and C# object types

```
public static readonly Dictionary<string, Type> PythonToC =
        new Dictionary<string, Type>
{
        {"uint8", typeof(byte)},
        {"int8", typeof(sbyte)},
        {"uint16", typeof(ushort)},
        {"int16", typeof(short)},
        {"uint32", typeof(uint)},
        {"int32", typeof(int)},
        {"double", typeof(double)},
        {"float", typeof(float)},
        {"time", typeof(TimeSpan)},
        {"id", typeof(int)},
        {"char", typeof(char[])},
        {"latitude", typeof(double)},
        {"longitude", typeof(double)},
        {"altitude", typeof(float)},
        {"speed", typeof(float)},
};
```

4.2.2 Code Examples

The IDD was the first section to be completed as we needed a way to maintain the IDD, and there was no way to do that before DocTest was in use. The back-end that was created was written in Python. This was a pet project of one of the developers that became useful so it was adopted to be used to create code and define the unit types. However, Python types and C# types do not easily match. The solution was to incorporate both types into each field within the IDD. The Python field is stored as the main type, but the C# field is used to determine the correct parser to use in determining the correct value. In Figure 10 you can see the dictionary that is used to transition between these two codebases. A dictionary in C# is used like a hash lookup where whatever key you give it you get at most one object back, in this case you get a type back.

Loading the specification files is very easy. All that needs to be done is to find each file that needs to be opened and then load it. This is done in two steps to allow for us to correctly update the progress, otherwise we would not know how many files were going to

be opened and would not be able to update the progress to discrete values. This can be seen in Figure 11, and it shows that the two step open and load allows for the progress to be updated nicely.

Figure 11: Code showing the loading of the IDD files and the updating of the progress.

```
Splash.Loading("Updating IDD Specification Files", 0);

string[] files = Directory.GetFiles(_window.IDDLocation);
foreach (string file in files)
{
        string[] fileAr = file.Split('.');
        if (fileAr[fileAr.Length - 1] == "spec")
        {
                IDDSpecFile spec = new IDDSpecFile(_window, file);
                _files.Add(spec);
        }
}

Splash.Loading("Loading IDD File" + (_files.Count > 1 ? "s" : ""),
         5);

for (int i = 0; i < _files.Count; i++)
{
        Splash.Loading("Loading " + _files[i].ShortName,
                (((double)i / _files.Count) * 40) + 10);

        _files[i].Load();
}

Splash.Loading("Done Loading IDD File" +
        (_files.Count > 1 ? "s" : ""), 50);
```

4.2.3 How to Make the IDD

The main window for the IDD, when opened (Figure 12), has the ability to add, remove and edit IDD files and their messages. On the top right the controls can be used to add a new IDD file. This file must contain one or more STANAG messages. When a message is created or already exists it is added to a list to make sure there are no duplicate messages. This used to be one of the major issues in the past with our previous IDD that had many duplicates and caused confusion on which message was valid. The list of messages in each file can be sorted however the user prefers. However, this sort has no affect on the sort in the IDD document that is generated. At the bottom are all the option controls that are available for the selected item and other buttons that are always active like "Create IDD" and "Create New Message". Double clicking on a message or selecting a message and then clicking on "Edit Msg" will open a new window to allow modifying of the selected message.

In Figure 13, the edit message window shows the various parts of the message. In the message shown it has 4 fields, with each field's name and type shown. There is also the message details that help the person looking at the IDD to know what the message is used for. The message's name and number are located at the top and have green checkboxes next to them. This indiciates that the message number and name are unique and can be used. If a number or name were used already there would be red "X's" instead showing there was a problem and would not allow the user to save or continue without fixing the mistake. Each of these fields can then also be modified by double clicking on them.

Each definition can be easily edited through this window, shown in Figure 14. At the top there are the basics of the message, what the field should be named, and what units are in the field. The units can be useful, as mentioned previously, because SI units sometimes do not pack easily into smaller fields and would take more bandwidth than necessary. The type can then be set and, if there are many of the same type, the count can be increased from 1. This is very useful when sending strings as they are simply an array of characters and need to have a finite boundary for sending over STANAG. The last 2 fields are merely for documentation and show what the expected maximum and minimum values are; there is even a button that will set the default max and min for the type selected. Longitude is set up to default to radians and it's maximum is π and its minimum is $-\pi$.

Figure 12: Main window for IDD List.

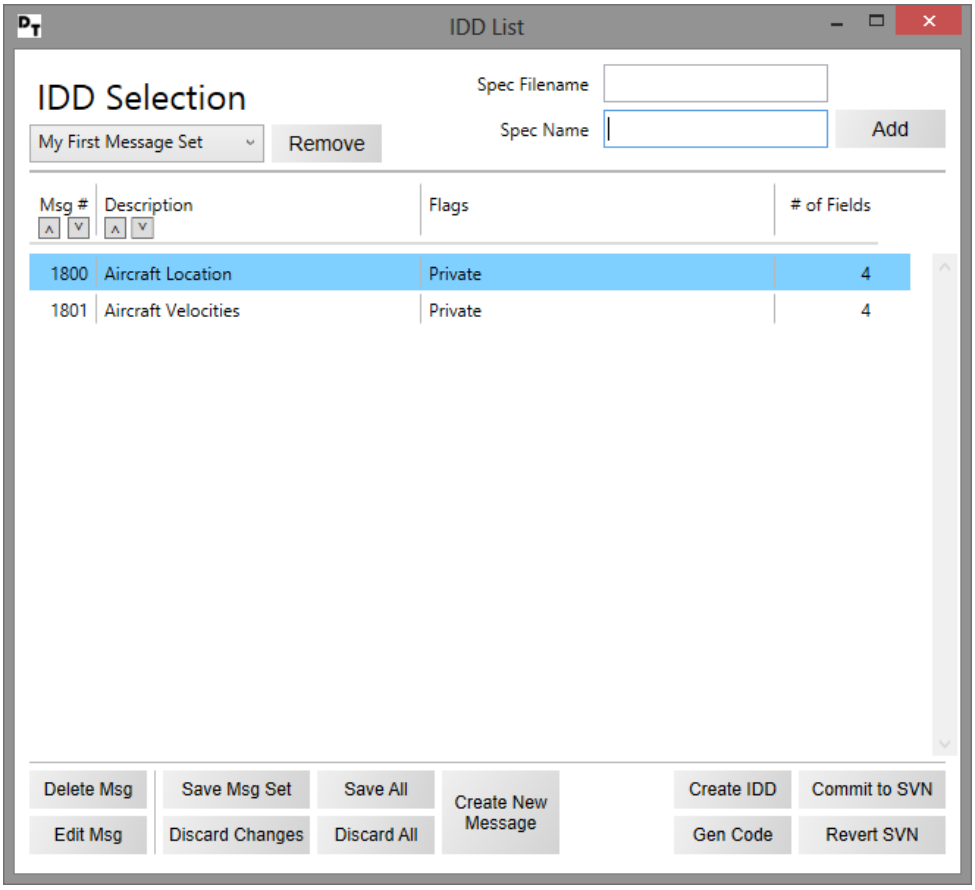


Figure 13: IDD Edit Message window that allows manipulation of a STANAG message.

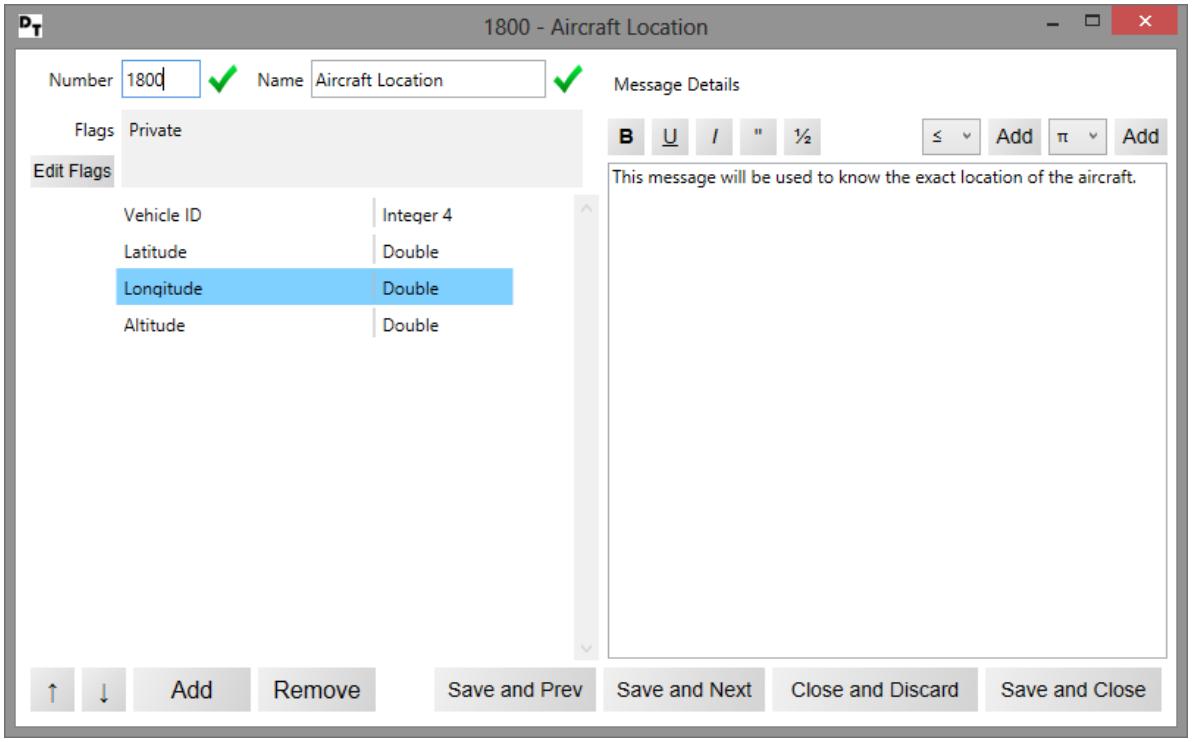


Figure 14: IDD window that allows for editing an individual definition within a STANAG message.

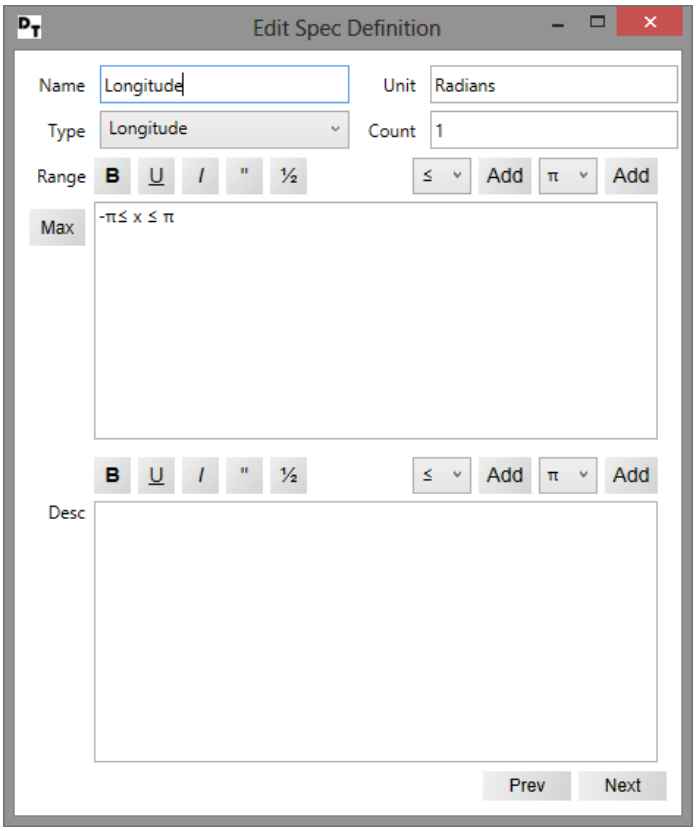


Figure 15: Example output of the IDD.

Message #1800 - Aircraft Location
This message will be used to know the exact location of the aircraft.

Table 6: Message #1800 - Aircraft Location

Unique ID	Name and Description	Type	Units	Range
1800.01	**Vehicle ID**	Integer 4	None	See Section 5.1
1800.02	**Latitude**	Double	Radians	$-\pi/2 \leq x \leq \pi/2$
1800.03	**Longitude**	Double	Radians	$-\pi \leq x \leq \pi$
1800.04	**Altitude**	Double	Meters	$-1{,}000 \leq x \leq 100{,}000$

Message #1801 - Aircraft Velocities
This message will give us the velocities of the aircraft.

Table 7: Message #1801 - Aircraft Velocities

Unique ID	Name and Description	Type	Units	Range
1801.01	**Vehicle ID**	Integer 4	None	See Section 5.1
1801.02	**Velocity North**	Float	Meters / Second	N/A
1801.03	**Velocity East**	Float	Meters / Second	N/A
1801.04	**Velocity Down**	Float	Meters / Second	N/A

4.2.4 Sample Output

The output of the IDD on the first run takes several minutes as there are many packages it requires to build all the functionality into the document. Each run after that only takes a few seconds as it just needs to compile the document and not download the packages again. Note that the lines do not show well in some PDF viewers, and thus capturing this causes it to not show correctly. A sample IDD Specification file was used to create the output in Figure 15.

4.3 DTP

The DTPs, Detailed Test Procedures, are used to have test procedures that capture as many bugs and issues as possible. If the software contains issues that are deemed too dangerous to use, they must be fixed or a work-around created. Some issues may be as simple as incorrect spelling, and some may be as serious as to cause the vehicle to not respond. It would be very bad and potentially expensive if software were to cause a mishap. Testing is the only way to verify that issues are not present in the software. The DTPs cover many tests and can be used in both SIL, Systems Integration Lab, and HITL, Hardware in the Loop, testing. These two tests are very similar but are run by different people, and the SIL testing is done with the hardware on a table and the HITL testing is done with the vehicle. Issues are always found, but most of the time the issues are minor enough to allow the software to be used for operations. Each new release must still go through each test procedure to re-verify, no matter how small the code change is.

4.3.1 File Structure

The DTP is stored in specification files using JSON. JSON allows for easy cross-platform storage that can easily be created as an object in DocTest.

1. Name
2. Description
3. Sections
    (a) Name
    (b) Last Updated (the date this was last changed)
    (c) Description
    (d) Testers (who is testing this section)
    (e) Step
        i. State (is this done, failed, or passed)
        ii. Action (what needs to be done to complete this step)
        iii. other fields are here, but contain sensitive information
    (f) Flags (what the message connects with)
    (g) Resources (images used in the description)
4. RunCount (which test pass is this)
5. StartDate (when the test was started)
6. Required (is this test required or optional)

4.3.2 Code Examples

JSON files are very easy to parse when properly formatted. The IDD files were taken from a previous project. They had incorrect formatting that made it difficult to parse correctly. However, the DTP files were created explicitly for this application and were created according to the JSON spec. In Figure 16, there are all of the required functions for the DTP File: Load, Save, Add, and Remove. In the load function there is a sanity check to make sure the file exists, as it would cause exceptions if a file that did not exist were attempted to be loaded. Once the file is found it is opened and deserialized into a DTP file object. Deserialization is the process of parsing a file from a known format into an object. This object is then used to load the DTP file it got called from. Saving is even easier as it just takes the current DTP file object and serializes it out to the file. Serialization is the reverse of deserialization and involves taking an object and converting it to a format; JSON is the file that we are serializing to in DocTest. Adding and removing sections is very easy for the DTPs. All that needs to be done is to simply call the appropriate function on the list of sections for the DTP file.

Figure 16: DTP code to load and save each DTP file.

```
public void Load()
{
        if (File.Exists(_filename))
        {
                StreamReader reader = new StreamReader(_filename);
                DTPSpecFile obj = JsonConvert.DeserializeObject
                        <DTPSpecFile>(reader.ReadToEnd());

                Name = obj.Name;
                Details = obj.Details;
                Sections = obj.Sections;

                reader.Close();
        }
}

public void Save()
{
        string json = JsonConvert.SerializeObject(this,
                Formatting.Indented);
        StreamWriter writer = new StreamWriter(_filename);

        writer.Write(json);
        writer.Close();
}

public void Add(ISpecItem section)
{
        Sections.Add((DTPSection)section);
}

public void Remove(ISpecItem section)
{
        Sections.Remove((DTPSection)section);
}
```

4.3.3 How to Make a DTP

Adding and editing DTPs is a straight forward process, similar to how edits ere done for the IDD. In Figure 17, the first window contains a drop-down menu of all the DTP files that are available. A new DTP file can be created using the top right textboxes. Once a file is selected, the individual sections are shown. In this case, "Starting Computer" and "Internet Connectivity" are the two sections already existing in the "Basic Computer Tests" DTP. Notice there are no details about this file, as the title is explanation enough. For other DTPs it might not be enough. When that is the case a developer can give a smart, concise explanation of what this DTP is supposed to accomplish. At the bottom of this window are all the options available, these controls will disable and enable based on what the user has selected. For instance, if no item was selected, the "Delete Item" would not be enabled as it would not have anything to delete. To edit a section you can either double click on the section in the list, or click "Edit Item".

This then brings up the DTP Section (Figure 18) that was selected in Figure 17. In this section control, each test can be sorted however the user prefers, adding and removing additional tests is as easy as clicking on "Add" or "Remove". Again this section can have more details if the tests are more obscure, which is the case for many tests that are performed by us. At the bottom of this window are buttons to control easy navigation. When making many changes it is nice to not have to keep changing windows, these buttons allow for quick maneuvering between each DTP section, and for closing without making modifications. To edit an individual test simply double click on it's corresponding control.

The last window in the DTPs is the individual test step window in Figure 19. This allows for the manipulation of all aspects of the test. It has each section that makes up the test and can be used to display the artifacts. In the example below, this step requires that a boolean be inputted when running the DTPs. This will make sure the user verifies the test and will also set the state of the test to passing if each artifact is correctly set or set to true. There are many more types that can be verified but they cannot be shown in this paper.

Figure 17: DTP List window that allows the addition of new DTPs or the editing of existing DTPs.

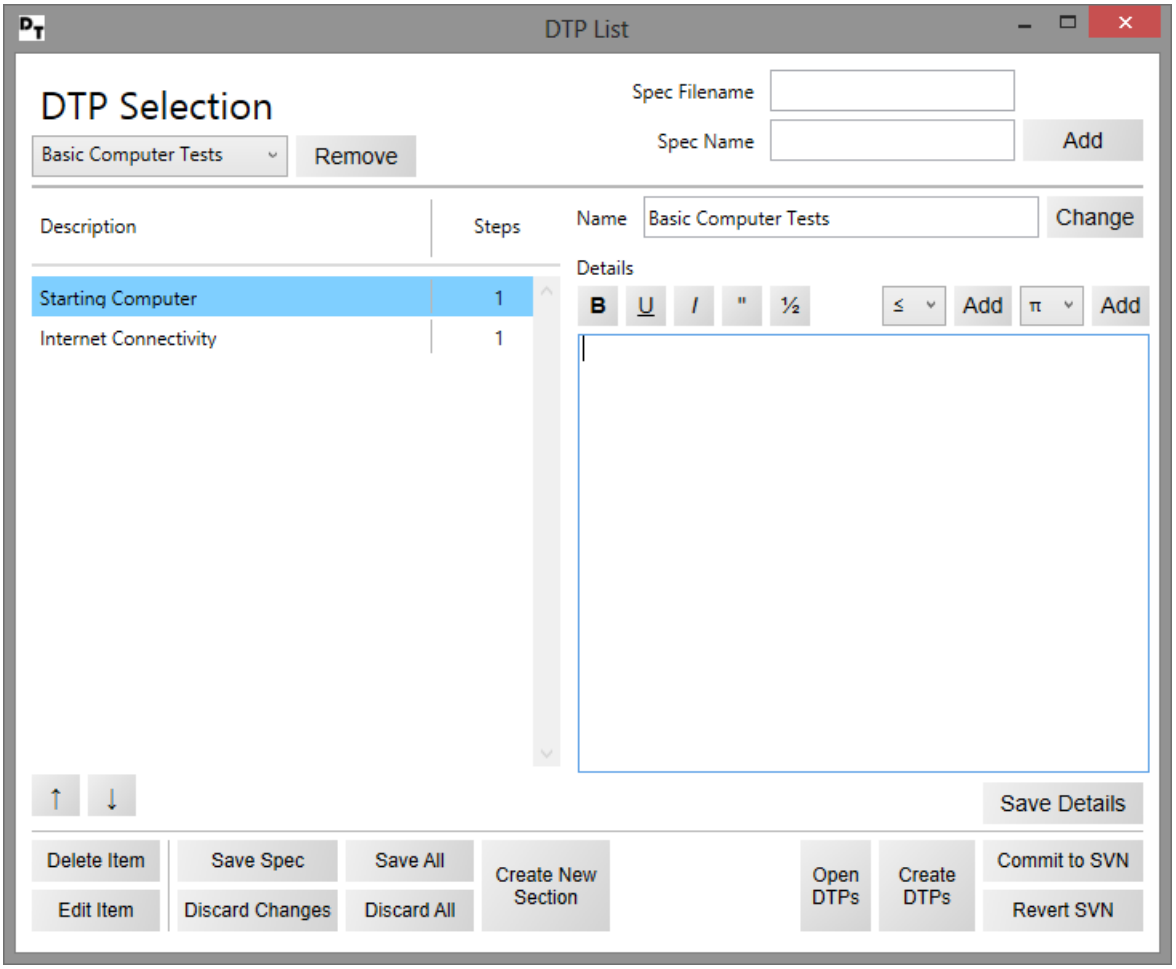


Figure 18: DTP Section window that allows the addition of new test items or the editing of existing test items.

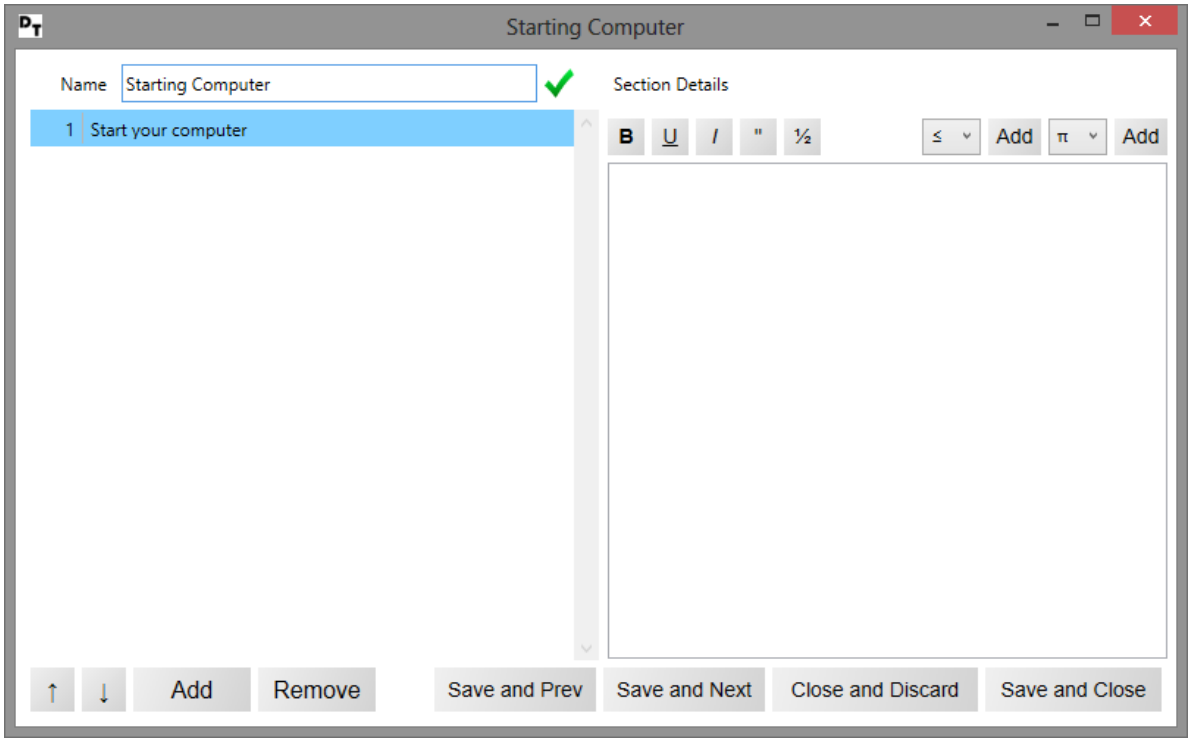


Figure 19: DTP Item window that shows the test item with each field that needs to be set to allow the user to test the desired functionality correctly.

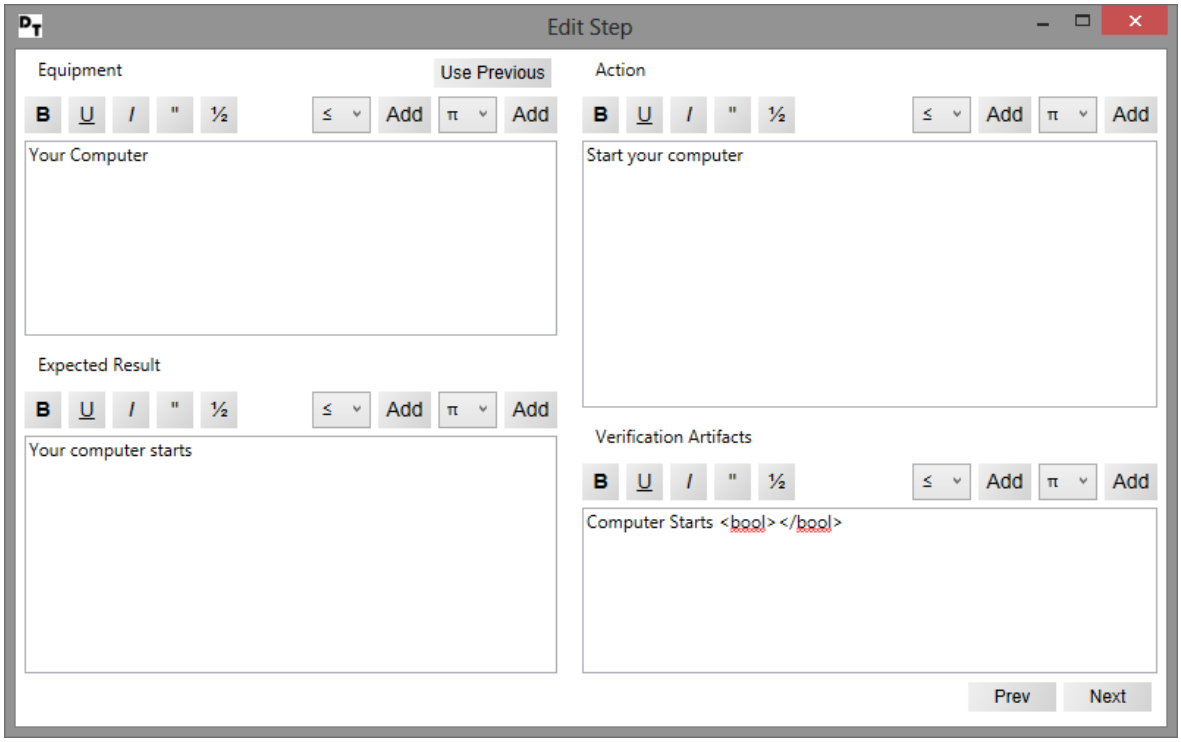
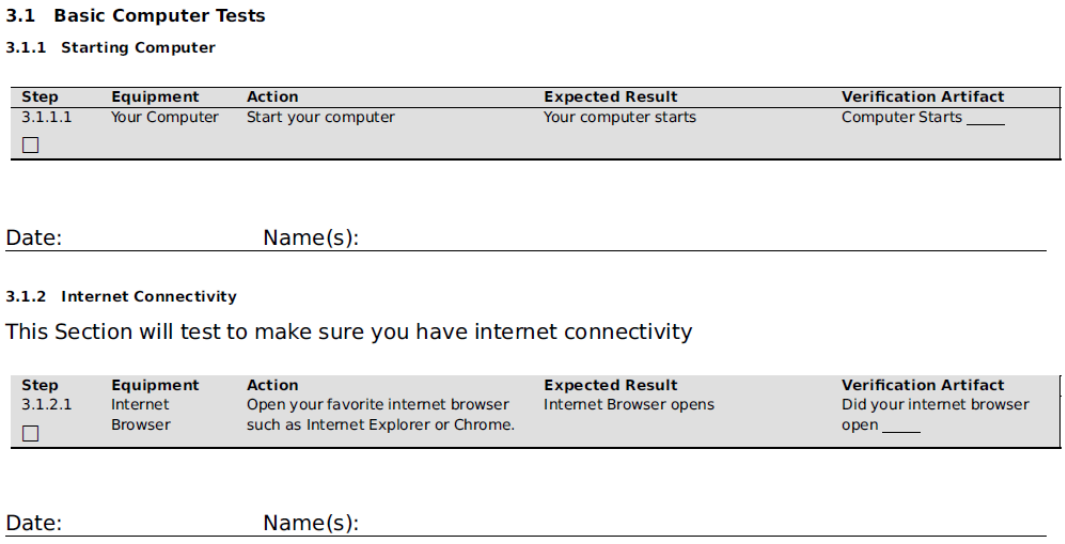


Figure 20: Sample output of the DTP document.



3.1 Basic Computer Tests

3.1.1 Starting Computer

Step	Equipment	Action	Expected Result	Verification Artifact
3.1.1.1 ☐	Your Computer	Start your computer	Your computer starts	Computer Starts ____

Date: Name(s):

3.1.2 Internet Connectivity

This Section will test to make sure you have internet connectivity

Step	Equipment	Action	Expected Result	Verification Artifact
3.1.2.1 ☐	Internet Browser	Open your favorite internet browser such as Internet Explorer or Chrome.	Internet Browser opens	Did your internet browser open ____

Date: Name(s):

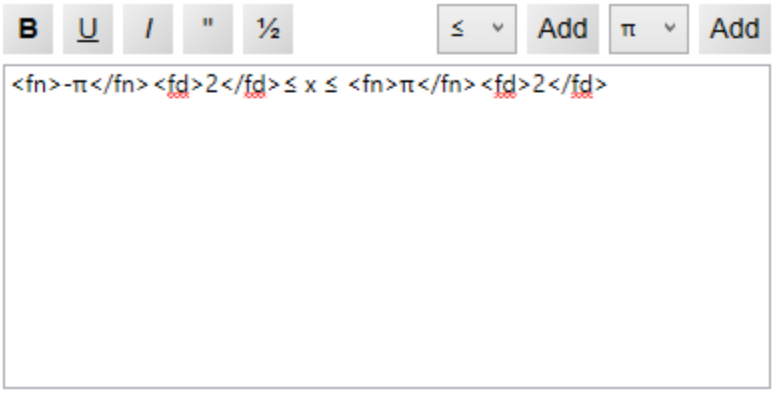
4.3.4 Sample Output

As with the IDD, the first run can take some time to download all the additional packages. The LaTeX files do share a great deal of the same packages and may take less time to compile after the initial run. Once compiled, the document can be used to run the tests, however, this application was designed mostly to be used in the testing environment and as such creating a document without the testing completed is just for testing purposes. Unfortunately I cannot show what a completed test DTP looks like as Lockheed does not wish to divulge this information. It does look similar to the sample output in Figure 20. While these fake tests are short, they show that each section has its own area to record the date when finished and who tested each section. There are also checkboxes for each test to make sure the user has completed each step. Also of note, the DTP document is created in landscape orientation to allow for more room for each test item, as there are several columns that need the additional horizontal space.

4.4 LaTeX Utilities

While creating DocTest, many things to create the LaTeX document were being duplicated. Functions to create tables and convert to LaTeX were both in the IDD and DTP. I then decided to create a small library that kept many of these functions in one place and could be easily accessed by each part of DocTest. One very important part of this was to create a

Figure 21: LaTeX Editor control that was created to make it easy for developers to input text.



document editor that could easily add commands that would be easily converted to LaTeX but still be user readable. I created a TextBox (See Figure 21) that allows the user to copy from Microsoft Word© [7] or enter their own text in user readable characters. One easily viewed example was when the description contained π. The user would have needed to type, "π", in LaTeX; instead this textbox and utilities package allows the user to copy and paste or select from a drop down. This allows the user to easily see what the output will look like while also translating correctly to LaTeX style commands.

LaTeX is an interesting language that takes some use to get a good grasp of. When creating a LaTeX file that has referencing, i.e. has page numbers or references items on other pages like figures and tables, LaTeX must be run twice. This is because the page must first be rendered then in the next run the referencing can be completed. Some very complicated features sometimes require more runs, however that is not within the scope of this application. In addition to these multiple runs, LaTeX requires the addition of many packages when more functionality it desired. The IDD contains 18 additional packages (Figure 22) that were needed to support various features, like multi-page tables and colored table cells.

In creating each table, there is a great deal of information that must be added to control the top of the table, the bottom of the table, and the page transitions to show when the table spans multiple pages. In Figure 23, each section can be seen. There is a caption that is shown for each table with the item's number and description. On the next line the top of the table, the header, is shown and has the ID, name and description, type, units, and range. If the table spans multiple pages then the table will also place the secondary header and primary footer. When the entire table is printed it then finishes by placing the last

Figure 22: LaTeX code that shows all additional packages required for the IDD.

```
\usepackage{array}
\usepackage{setspace}
\usepackage{url}
\usepackage{geometry}
\usepackage[pdftex]{graphicx}
\usepackage{longtable}
\usepackage{hyperref}
\usepackage{fancyhdr}
\usepackage{lastpage}
\usepackage[table]{xcolor}
\usepackage{fixltx2e}
\usepackage{float}
\usepackage{supertabular}
\usepackage{xfrac}
\usepackage[none]{hyphenat}
\usepackage{ragged2e}
\usepackage[T1]{fontenc}
\usepackage[scaled]{berasans}
```

Figure 23: Code to create a LaTeX "longtable" for the IDD

```
writer.WriteLine(@"\normalsize{");
writer.WriteLine(@"\begin{center}\rowcolors{2}{tableShade}{} ");
writer.WriteLine(@"\begin{longtable}{ | p{1.8cm} | p{3.6cm} |
        p{2cm} | p{2cm} | p{3.6cm} |}");
writer.WriteLine(@"\caption{Message \#" + item.Number + " - " +
        item.Description + @"} \\ \hline");
writer.WriteLine(@"\hline \textbf{Unique ID} & " +
        @"\textbf{Name and \newline Description} & " +
        @"\textbf{Type} & \textbf{Units} & \textbf{Range} " +
        @"\\ \hline");
writer.WriteLine(@"\endfirsthead \hline");
writer.WriteLine(@"\multicolumn{5}{|l|}");
writer.WriteLine(@"{Continued from previous page} \\");
writer.WriteLine(@"\hline \textbf{Unique ID} & " +
        @"\textbf{Name and \newline Description} & " +
        \textbf{Type} & \textbf{Units} & " +
        @"\textbf{Range} \\ \hline");
writer.WriteLine(@"\endhead");
writer.WriteLine(@"\hline \multicolumn{5}{|r|}{{Continued " +
        @"on next page}} \\ \hline");
writer.WriteLine(@"\endfoot");
writer.WriteLine(@"\hline");
writer.WriteLine(@"\endlastfoot");
```

footer, which in this example is simply a black line.

4.5 Interface Design

As stated previously, when the design was expanded there was a requirement to allow for easy additions to the program. This required a change in code to allow for this. Interfaces were designed to capture the similar aspects of each component to allow for less or no dependability on code for each document type. Although not all document references have been removed yet, a great majority have been simplified. In Figure 24, the interface contains many functions each of which were required for both the DTPs and the IDD. This interface also references other interfaces including ISpecFile (Figure 25), ISpecEditor, and ISpecControl. The last two, ISpecEditor and ISpecControl, are only used to differentiate between objects and do not contain any interface design inside of them. These two interfaces are empty interfaces, They are only used to keep their respective classes that they are associated with separate. By keeping these objects seperate, it allows for the managers to keep track of each object that is associated with each interface. Each interface name is preceded by an "I", this is a standard and allows for programmers to easily note what is a class file and what is a interface file.

ISpecFile (Figure 25) also contains another interface called ISpecItem. This reference is only in name, as the interface contains nothing as there were no similarities between the DTP and IDD items, at least none that were of value to create into the interface. This interface is very clean, having only 4 required functions. Each file needs to have the ability to load and save itself, and add or remove items from itself. The manager calls all necessary functions. For instance, when the program is loading, each manager calls the "Load" function that each of the files have. This loads all the necessary components for each file, including all of it's items.

Figure 24: Interface code that is used to abstract the document classes to allow for easy program additions.

```
public interface ISpecManager
{
        void IntializeControls();
        ISpecItem CreateNewItem(ISpecFile file);
        void ReloadFile(string filename);
        void ReloadFile(ISpecFile file);
        void ReloadAllFiles();
        void ShowEditor(ISpecControl ctrl);
        void GetNextMsgEditor(ISpecEditor editor);
        void GetPreviousMsgEditor(ISpecEditor editor);
        void AddControl(ISpecControl ctrl);
        void RemoveControl(ISpecControl ctrl);
        void RemoveFile(ISpecFile file, bool force);
        void AddFile(ISpecFile file);
        void AddFile(ISpecFile file, bool save, bool addToSVN);
        void SaveFile(ISpecFile file, bool forceSave);
        void SaveAllFiles();
        ISpecFile GetSpecFile(string specName);
        ISpecFile GetSpecFile(string specName, bool getFromHDD);
        void CommitSVN();
        void RevertSVN();
}
```

Figure 25: Interface code that is used to abstract the document classes to allow for easy program additions.

```
public interface ISpecFile
{
        void Load();
        void Save();
        void Add(ISpecItem item);
        void Remove(ISpecItem item);
}
```

5 Results

We have been testing this application for quite some time now. It has been in use since it was first created. Along the way we found issues and added features; both of which take time and money. There have been many talks of how Lockheed Martin can improve their documentation, DocTest is actually able to do that while also lessening the workload of each developer.

5.1 Cost and Time

Initially this program was to take around 90 man hours. More scope was added after only 50 hours of work being put into the application. We were given around 1.5 man months of time to get all the necessary additions into DocTest. I then spent an entire month working on DocTest to make it a better program for us than what was initially planned. Another coworker also put in around 2 man weeks to help input the data and test the application. In total DocTest cost 290 hours of work, 210 directly from me. With the average software engineer making $68,548 according to PayScale.com[8], we can see that DocTest cost almost $10,000 to create.

This is a very large investment into automating our documentation and testing procedures when considering this started out as a small senior project. Many large companies have been trying to enhance their documentation both for their own record tracking and for possible audits from contracts. If documentation is sparse or inaccurate a contract can be voided if found to be negligent in court. If keeping our documentation up-to-date and correct will help us in not becoming negligent then $10,000 is a fairly small price to pay. Both AS-9100[3] and ISO-9001[2] require strict guidelines that must be followed to receive and maintain certification. These standards are set in place to show when a company has a good set of documentation. These pertain mostly to quality of our product, but each aspect of the development sheds light on the final product. AS-9100[3] sets up strict requirements to show how quality is maintained. This takes it a step further than ISO-9100[2] because of the high-risk nature of aerospace. ISO-9001[2] sets up a series of principles that should be followed to maintain quality and also requires a series of internal auditing to verify procedures are being followed. Many companies hire external reviewers to get a more impartial review of the process.

5.2 Quick Facts

- Line Count: 10,566 lines of code
- Executable Size: 706KB
- Source Size: 3.05MB
- Hours To Design and Program: 180 Hours
- Hours Testing: 110 Hours
- Hours Total: 290 Hours

6 Future Possibilities

When the scope of DocTest was increased I was asked to develop the application to be able to be easily added to in the future. The easiest code style would have been simply making all the necessary classes to make DocTest work for what it was intended. However, an interface had to be created to allow for other similar additional classes in the future. Resources needed to be shared without having to have knowledge of each part of the application. Having parts of the code separated by interfaces allows for easy changes to each part without affecting the others. This way any developer could more easily learn the structure and create extensions to the default behavior. Currently more work has been added to this application. This enhances Lockheed Martin's automated document creation to further remove the possibility of user error in editing these documents by hand, or in some cases adding documentation where none existed in the first place.

7 Summary

This project started out being a straight forward senior project but evolved into a large application that has already been deployed, and is now scheduled to grow even more. The requirements that were started out with changed with the availability of time and made a program that was much better than was first envisioned. Currently, more functionality has been added to this application to again enhance our automated document creation. This helps further remove the possibility of user error in editing these documents by hand or in some cases adding documentation where none existed in the first place. DocTest has already made a large impact at Lockheed Martin UIS and it will only help us further our goals at having great documentation that supports Lockheed Martin's programs.

References

[1] Adobe, "About adobe pdf," March 2013. [Online]. Available: http://www.adobe.com/products/acrobat/adobepdf.html

This is the format that LaTeX saves to, it is widely used throughout the world and has many great features that lends itself to professional quality documents.

[2] ISO, "Iso 9000 - quality management," March 2013. [Online]. Available: http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm

[3] ISOQAR, "Isoqar - as 9100 the standard for aerospace," March 2013. [Online]. Available: http://www.alcumusgroup.com/isoqar/industry-standards/as-9100/

[4] LaTeX, "Latex - a document preparation system," March 2013. [Online]. Available: http://www.latex-project.org/

This is the format that is used to generate the PDF documents; it is how DocTest compiles the data into a PDF file.

[5] L. Martin, "Lockheed martin," March 2013. [Online]. Available: http://www.lockheedmartin.com/

The main site for Lockheed Martin. This is where DocTest will be used to better our documentation procedures.

[6] Microsoft, "Microsoft directx," March 2013. [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=35

This is Microsoft DirectX and comes pre-installed in any Microsoft Windows computer.

[7] ——, "Microsoft word," March 2013. [Online]. Available: http://office.microsoft.com/en-us/word/

This is used to show the vast differences between LaTeX and Microsoft Word.

[8] PayScale.com, "Software engineer salary," March 2013. [Online]. Available: http://www.payscale.com/research/US/Job=Software_Engineer_%2F_Developer_%2F_Programmer/Salary

This site shows the average wage of almost any position available, in this case this shows the average wage of a Software Engineer.