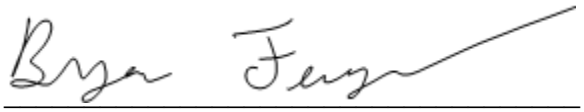
2009

DADiSP & SIMULINK Integration

A Major Qualifying Project
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the
Degree of Bachelor of Science

by



Bryan Ferguson

Date:
5 May, 2009

Advisor:
Hossein Hakim

ABSTRACT

MATLAB with Simulink is a powerful application set for signals processing in real-time or offline however can be slow to learn and is unattractive. DADiSP is menu driven through a colorful, intuitive Graphical User Interface but has limited signals processing capabilities and only processes data offline. This project sets out to conjoin these applications using Microsoft ActiveX technology to utilize the best features from each resulting in an easy to use and powerful real-time signals processing application with eye-catching outputs.

List of Contents

List of Figures

1. Introduction

Society of 2009 is based heavily on the concept of ubiquity. Everything from Dunkin Donuts and McDonalds to Microsoft Office, Cloud Computing, Smartphone Devices and Facebook. Everyone is a user of technology while technology and computing is now embedded in almost every minute of our lives. More and more often, users want to customize and program their devices to fit their data, needs and lives. To that end the ability to program and customize a particular application has become almost as ubiquitous as the application itself. This makes sense for engineering and research applications such as Maple or LabVIEW, but the ability to automate even Microsoft Office applications such as Word and Excel through Visual Studio Tools is a testament to our need to customize.

Some applications lend themselves to programmability more than others. Users today expect applications to be attractive. They expect a Graphical User Interface (GUI). All the while expecting it to be simple to use, yet perform complex functions, arguing as well that these two concepts are not mutually exclusive. Companies like Microsoft and Apple have implemented frameworks within their operating systems to allow independent software vendors (ISV) to be able to open up their applications up to end users without opening all of their source code. Such technologies are known as AppleScript, Component Object Model (COM), ActiveX and .NET. Microsoft has even gone so far as implementing ActiveX on Mac OS.

Technologies give the end user and application programming interface, or API. It's not as useful as having the whole source code, but it does provide a structured universe to interface between applications. The only problem with this implementation is that it's up to the ISV as to how to build the API. The ISV decides what functions, commands and data are available through the API. The ISV can also decide not to make an API available at all.

The documentation of the functions as well as how they work is also up to the vendor to supply. More importantly than the API being available, is documentation of the API and the API's functions. An ISV can build the most powerful API, but if there is no documentation for how to use it, then it is useless.

Because APIs can be so general, there are a number of ways to use them to communicate, which is completely dependent on the API implementation. For this reason it is up to the implementer to decide how to utilize the API. Some applications will allow direct connection to another, such as a client/server implementation where one application (client) makes a direct connection to another (server). Or some applications can only be classified as a server, so a 3rd application needs to be brought in as a broker to pass data and commands back and forth.

Two applications that have these types of APIs are MATLAB from the MathWorks and DADiSP from DSP Development Corporation. Both of these applications are commonly used for digital signals processing and both also incorporate an implementation of a Microsoft ActiveX API. While these are two applications that are marketed towards a similar goal, they differ in their implementations and functions.

MATLAB was developed in the 1970s and is short for Matrix Laboratory. It is an evolution of the linear algebra packages LINPACK and EISPACK which have a foundation in the FORTRAN programming language. MATLAB is command based, similar to a windows command line or a UNIX terminal. This lends itself to batch style scripting through the proprietary M-Files. Unfortunately this doesn't mean MATLAB is the most attractive application to use out of the box.

MATLAB does however have an ActiveX API. This API allows end users to control MATLAB as an ActiveX Server or also use MATLAB as an ActiveX Client to control another ActiveX Server. This makes it very versatile, however in order to use it as an ActiveX client, the coding must be done using proprietary

M-Files which are based on FORTRAN. This can be a positive or a negative depending on the point of view of the designer.

MATLAB later introduced an add-on package called SIMULINK that is a block diagramming and real-time execution environment. SIMULINK has block sets, some are standard and others can be purchased for a fee. These block sets are used for specific functions and applications within the SIMULINK environment. One such set of blocks allows users to implement code they have written in the form of an M-File and reconfigure it so that it will run in the SIMULINK. This has major implications in that all of the power of the ActiveX client connection and MATLAB's integrated development environment under the control of a block diagram based real-time environment.

Meanwhile, DSP Development Corporation was founded in 1984 with the concept of graphical data analysis and eventually produced an application called DADiSP. DADiSP is a worksheet application, similar to Microsoft Excel. Instead of putting simple numbers or simple mathematical formulas in each cell, users instead can enter tables of information, such as the array that comprises a signal, like that of a wav file. The data is represented in a graphical form instead of numerical form, allowing the user to see the data.

Like Microsoft Excel, other cells, or windows, can be dependent upon the first cell. Instead of performing simple mathematical calculations on a small set of numbers, DADiSP allows users to manipulate large series of data and perform complex calculations such as Fourier transforms and digital signal filtering. DADiSP utilizes a graphical user interface for almost all tasks, which makes it appealing to users that are not as comfortable with computer programming languages.

DADiSP does not, however, have any sort of real-time environment. All tasks in DADiSP are asynchronous. It also does not have any sort of block diagramming features. This limits its usage for real time experiments. DADiSP does implement an ActiveX Server and Client API and also has its own

proprietary scripting language called SPL, based in C/C++. This scripting language is not only useful to script within DADiSP, but all of the SPL commands and utilities are available through the ActiveX Server implementation meaning they are available to be executed from a remote application.

The purpose of this project is to provide a sample SIMULINK model that can be installed via a standard Microsoft installer (MSI) package. This package is a proof of concept to show the feasibility of utilizing DADiSP with real-time and block diagramming environments. The model utilizes ActiveX connections to pass data between MATLAB as the ActiveX client and DADiSP as the ActiveX server. This model offers the block diagram visualization and real-time power of SIMULINK and at the same time the directly paged virtual data series and more attractive and flexible graphical interface of DADiSP. This also allows for the end user to perform post manipulation in DADiSP after the real-time execution is complete, which gives the user full access to the DADiSP tools suite, as opposed to only the ones that are activated by the SIMULINK model and chosen by the developer.

In MatLab and SimuLink, a data set cannot be manipulated unless the entire set is loaded into RAM. While RAM is less expensive than it was many years ago, it still has generally a much smaller storage capacity than a hard drive and the pagefile. DADiSP allows for directly paged virtual data series which reside within the page file, or scratch space, on a hard disk as opposed to being loaded into the limited space in RAM. Data sets can also be paged directly by the DADiSP application as opposed to using operating system page file algorithms which can add lag time to page reading and writing. All of the execution and paging is completed by DADiSP automatically and seamlessly to the user.

In MatLab, the GUI has much to be desired, in reality it is command prompt with a number of toolbar surrounding it. The graphs produced by MatLab are static and can only be manipulated, rescaled or given titles through the command line that was previously discussed as lack-luster in visuals. In DADiSP, all of the Graphs and Charts can be tweaked and updated to look presentable without having

to touch a command prompt. Axis can be labeled and adjusted and just about anything desired can be performed through a “What You See Is What You Get” (WYSIWYG), pronounced whizzywig, interface, which is simpler and faster to learn and us.

2. Background

In order to understand how to build on and enhance two existing technologies, we first need to understand how all the technologies involved will work. This means we must understand where DADiSP and MATLAB came from and how they have evolved. However we have also added in the aspect of ActiveX, an integration technology and standard from Microsoft.

This section will discuss the history of ActiveX and the history behind it; including the ActiveX roots in the Component Object Model (COM). An addition this section will discuss the future for the Component Object Model in Microsoft .NET Framework and the evolution of the technology. This will also discuss the past and evolution of DADiSP, MATLAB and SimuLink as well as how the ActiveX framework is utilized within their environments.

2.1 MATLAB

MatLab is an operating environment for Data analysis and numeric computation using matrices. MatLab can use FORTRAN style interpreted code for scripting applications and is also constructed in FORTRAN. *("MATLAB - Introduction and Key Features")* SimuLink is an application that is integrated with MatLab. Simulink is an environment for simulation and modeling. Simulink, in simple terms can be seen as a block diagram application. Since MatLab and Simulink are tightly integrated, SimuLink has the scripting and computation functionality of MatLab underlying it. This produces can produce power solutions for engineering applications. *("Simulink - Introduction and Key Features")*

2.1.1 MATHWORKS History

Between 1972 and 1973 Cleve Moler joined a project to take some libraries developed by John Wilkinson in ALGOL designed to determine eigenvalues numerically, and port them to FORTRAN. This package was called EISPACK. In 1978, Cleve worked with Jim Bunch, Jack Dongarra and Pete Stewart to write a follow-up, LINPACK, which was a library designed for solving linear systems and equations numerically, this time written from the ground up in FORTRAN. *("Video: The Origins of MATLAB")*

Cleve wanted to allow his students to be able to access LINPACK and EISPACK without having to write FORTRAN. To do this, Moler used concept developed by Niklaus Wirth for a parser and implemented it on top of LINPACK and EISPACK and called it MATLAB. Matrices were the only data type allowed. The entire program comprised approximately 2000 lines of code and 80 different functions compared to the 8000 different functions of today. *("Video: The Origins of MATLAB")*

In 1979 Cleve spent a sabbatical at Stanford University, where he taught a computer science course in numerical analysis. Many of the numerical analysis students were not interested in MatLab. However many other engineering students in the course found MatLab useful for signals processing and control theory. Some of these students shared MatLab with Jack Little, a recent graduate from Stanford, who saw potential in the aforementioned areas. *("Video: The Origins of MATLAB")*

Together, Jack and Cleve founded The Mathworks in 1984. Jack co-authored some early versions of Matlab as well as the signal processing toolbox and the control system toolbox. Currently Jack Little is the President of The Mathworks and Cleve Moler is the Chief Scientist. *("The MathWorks - Founders - Jack Little")* The Corporate Headquarters for The Mathworks is in Natick, Massachusetts *("The MathWorks - Contact Us - Worldwide Offices and Representatives")*.

2.1.2 ActiveX integration and implementation

Version 5.0 of MatLab allowed ActiveX clients to utilize MatLab as an ActiveX Server. This allowed users to control MatLab via an external application. Such examples might be using a Microsoft Visual Basic program to send data to matlab and use that data and create embed a Matlab Graph or plot. However, in 1998 Mathworks released version 5.2, also known as MatLab release 10 that included support to now utilize MatLab as an ActiveX Client. Now Matlab can be used as an ActiveX Client to connect to ActiveX server applications using built in Fortran-like Code. *("MATLAB News & Notes - Summer 1998 - MATLAB Compiler and New Support for ActiveX in MATLAB 5.2").*

Using MatLab as an ActiveX client can be useful to now embed controls from other applications into MatLab. This is useful when MatLab doesn't provide a visual control or functionality for something a user needs. For example, users can make a connection to Excel and embed a calendar control from excel directly into a MatLab figure, as shown in Figure 1. *("MATLAB News & Notes - Summer 1998 - MATLAB Compiler and New Support for ActiveX in MATLAB 5.2")*

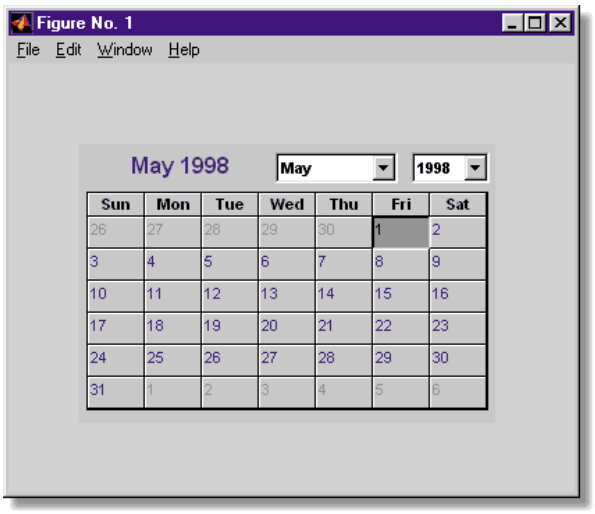


Figure 1 - Excel Calendar Embedded in MATLAB

Some applications that don't have controls that can be embedded like the calendar application above, but MatLab can still execute and show those applications in their own native process window. Depending on how a particular application is built, the ability for it to be embedded does not generally affect how it passes or handles data. For all intents and purposes, the above calendar application could be displayed through an excel window, yet still provide the same functionality, but it wouldn't look as clean or as unified as it does above. All of this functionality is taken advantage of through the use of M-Files and the FORTRAN based MatLab language. *("MATLAB News & Notes - Summer 1998 - MATLAB Compiler and New Support for ActiveX in MATLAB 5.2")*

2.1.3 MATLAB evolution and SIMULINK Implementation

Simulink is a model-based design and simulation environment that is distributed as part of MatLab, however it is licensed and sold as an add-on unit. Simulink is integrated with MatLab, which provides a powerful connection to the matrix based solutions as well as other add-on toolboxes provided by MatLab. However Simulink has its own library of functions. As Simulink is a model-based environment. All of its functions are organized in block sets and put together as a block diagram. Blocks can provide a distinct individual function, or they can also be a complex sub-system of their own with multiple inputs and outputs. Users can also design custom blocks of their own. *("Simulink - Introduction and Key Features")* Figure 2 shows an example of multiple embedded models within a master simulink model.

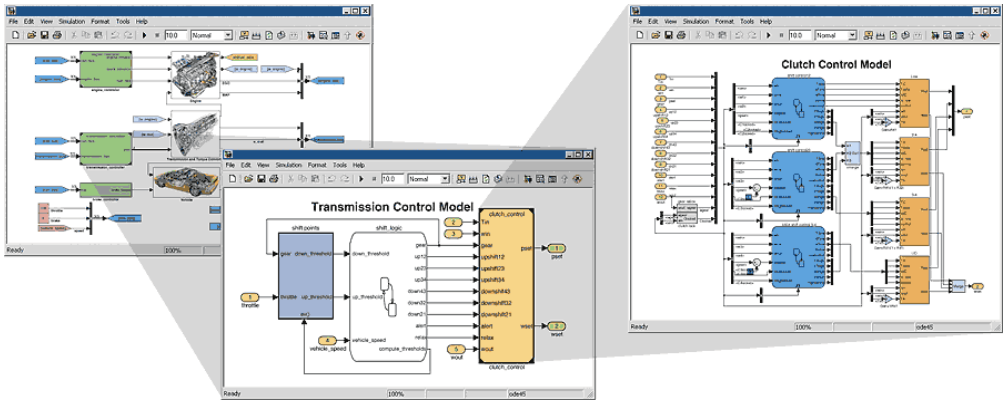


Figure 2 - Simulink System Example

Since Simulink is integrated with MatLab, and MatLab supports the ActiveX protocol, Simulink can reap the benefits of Matlab's interconnectivity with other ActiveX devices. Simulink can indirectly control ActiveX Servers using MatLab as an intermediary. This is done by using custom blocks that are either s-function blocks or MatLab Function blocks. S-Function Blocks can use specially crafted M-Files to define custom functionality and specify input and output. *("Simulink – Documentation")*

Figure 3 shows a Simulink Model with a few different types of customized blocks. The "From Wave File" block takes in a specified raw audio file in WAV format and converts it into the individually sampled bits. The S-Function block utilizes the input it's given out of the unbuffer from the WAV file and performs the actions specified by the parameters of the S-Function as shown in Figure 4.

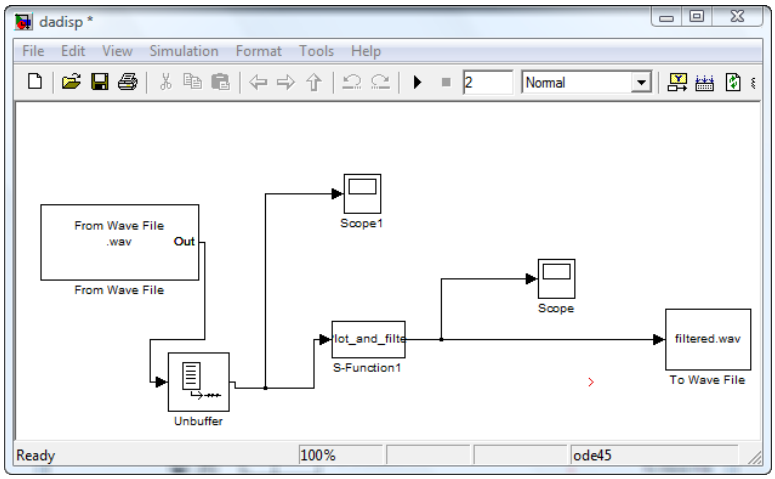


Figure 3 - Simulink Model with S-Function

Figure 4 shows the parameters window for the S-Function Block. The S-function name is the filename of the specially crafted M-File in the local directory of the model. The other S-function parameters are customized parameters by the creator or the M-File. These parameters are input by the user through the Parameters menu and can be passed between various functions of the M-File, as the M-File itself can contain many different functions. Is the S-Function references and uses additional files or modules, those can also be specified in this menu.

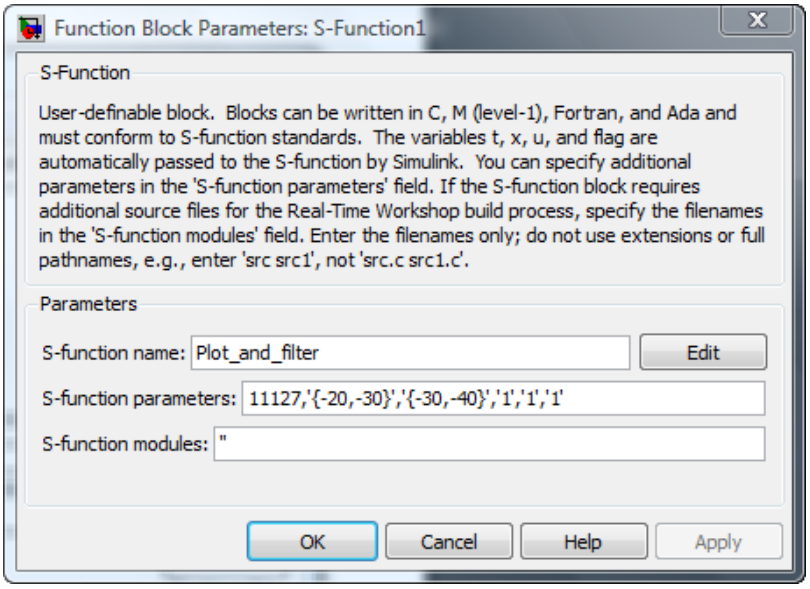


Figure 4 - S-Function Parameters

More information about how this specific S-Function and sample model work provided in section 3.1.4 Creating a Sample Model. This section will also describe what the specific parameters mean and walk through the underlying S-Function code line-by-line.

2.2 DADiSP

DADiSP is signals analysis software developed and distributed by DSP Development Corporation. DADiSP is designed as a signals processing spreadsheet, similar in function to Microsoft Excel. Each worksheet contains as many cells as deemed necessary by the user. Cells can contain raw data or

analyzed data based on other cells. If source data is updated, dependant cells update on the fly. Each cell doesn't just display numeric values, but rather the graphical representation of the raw, or analyzed data as described below. *("DSP Development Corporation: About DADiSP")*

> *"With DADiSP you can acquire, input and even generate sample data, displaying the results in multiple windows for immediate graphic comparison. You can reduce and transform data, using any of hundreds of menu-driven analysis functions, instantly graphing the results of each stage of your work. DADiSP doesn't require mastering any programming skills or arcane command sets. DADiSP was designed to let you perform data analysis the way you think about data analysis." ("DSP Development Corporation: About DADiSP")*

2.2.1 DADiSP Company History

DSP Development Corporation was founded in 1984 and currently holds their headquarters in Newton, MA. DSP Development supports such customers as NASA, General Motors, Boeing and Siemens. *("DADiP Executive Summary Flash 3 Presentation")* Their mission is to *"Develop high performance and reliable data analysis and display software that allows technical professionals who are not programmers to easily perform complex computations and visualization of their data in a familiar spreadsheet environment."("DSP Development Corporation: Corporate Background").*

DSP Development compares their product, DADiSP with that with high end engineering applications, but also markets it as a high end spreadsheet. Their goal is to provide a tool to scientists and engineers, similar in functionality to Microsoft Excel that does not require programming knowledge. This approach allows for users to manage large sets of data and visualize them in a meaningful way, as well as manipulate the data reliably. DSP Development markets DADiSP as a type of productivity application for engineers. *("DSP Development Corporation: Market Background")*

2.2.2 DADiSP Application Evolution

Like Many applications, DADiSP has evolved over time to become the application it is today. The current version of DADiSP is version 2002, marked by the year of its last full release, and however has continued to see improvements since. The owner of DSP Develoment Corporation, Randy Race, is also the sole developer of DADiSP. *("Race")*

DADiSP supports integration with certain data acquisition applications, user programming via Series Programming Language (SPL) and application integration via ActiveX. These integration and customization tools give end-users flexibility to utilize DADiSP in a multitude of ways. DADiSP can be used as a stand-alone application with separate data acquisition or with integrated acquisition. DADiSP can also be used in conjunction with any other application that is compatible with the ActiveX standard. *("DSP Development Corporation: DADiSP Product Family")*

While DADiSP supports full ActiveX support, it doesn't support individual window embedding. In 2008, DSP Development Corporation introduced a new product known as the DADiSP Application Builder (DAB). The DAB allows application developers to utilize functionality similar to the ActiveX implementation of the full version of DADiSP, however is in a lightweight redistributable package.

> *"Purchase a one-time license for each development machine and distribute your applications with no further fees or royalties. A callable install program is included to provide a single, integrated installation strategy for your target application deployment." ("DSP Development Corporation: DADiSP Application Builder")*

This new tools is the next logical step in utilizing DADiSP as a componentized piece of an integrated solution. It is a much smaller size and ideal for being embedded into custom applications. *("DSP Development Corporation: DADiSP Application Builder")*

Figure 5 displays a sample application developed by DSP Development Corporation to sample how one can use the DADiSP Application builder to embed individual pieces of DADiSP into a visual basic

program that provides a cleaner, unified look and feel to any custom built application. These could also be embedded in such applications as MatLab or SimuLink. Samples are also available to download from the DSP Development website. *("DSP Development Corporation: DADiSP Application Builder")*

Looking specifically at Figure 5, it shows 3 different DADiSP plots within one window that looks similar to a DADiSP window, however it is actually a custom-built application. The radio buttons, slider and buttons on the left hand side are all parts of the native custom-built application, but through ActiveX these controls and change the data or settings of the DADiSP graphs on the right to change the output. Building these controls in DADiSP is extremely difficult, however building them in Microsoft Visual Studio is fairly trivial because they are predefined controls through the development environment. *("Race")*

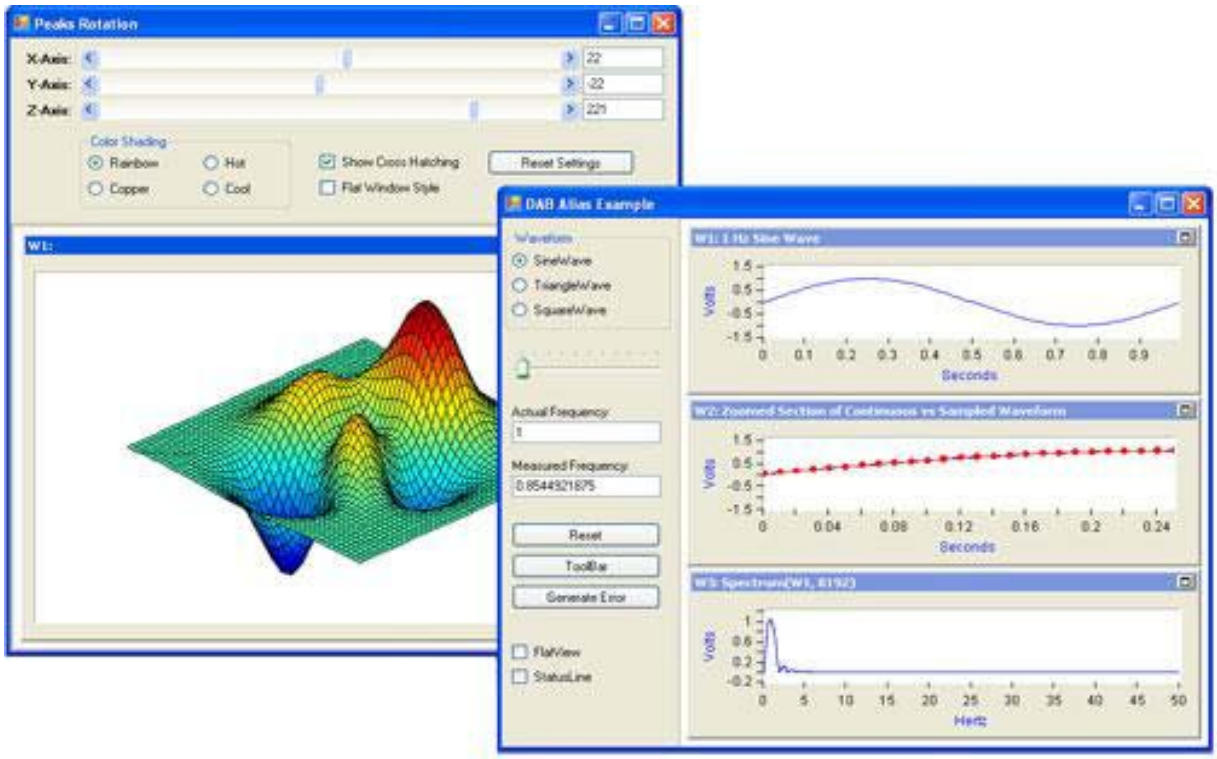


Figure 5 - DADiSP Application Builder Application

2.2.3 Integration of ActiveX and implementation

ActiveX is really nothing more than a set of standards as defined by Microsoft. ActiveX is described by the Component Object Model (COM) which is described in further detail in section 2.3. The idea behind COM, is that it's a way to interconnect processes independent of the language that processes use. Applications written in VB, C or a multitude of other languages can interact with one another. It also allows an implementer to utilize the functionality of an application without full knowledge of all the underlying logic. This prevents the implementer from reinventing the wheel. *("Component Object Model")*

As an ActiveX Client, DADiSP allows the end user to connect to other ActiveX enabled applications and utilize their functionality directly from with DADiSP using SPL. As an ActiveX server, other ActiveX clients can use their own native language to make a connection to DADiSP and utilize its functions. *("DSP Development Corporation: ActiveX")*

For Example, visual basic script (vbscript) can create an object that references the DADiSP application and makes a connection to the application in visual basic scripting language. Once connected, vbscript can insert and retrieve data as well as execute commands within DADiSP. This utilization of DADiSP as an ActiveX server is particularly convenient for use with MatLab and Simulink. *("DSP Development Corporation: ActiveX")*

When connecting DADiSP as an ActiveX server it supports three main methods: "GetData", "PutData" and "Execute". The "GetData" method will retrieve data from a DADiSP window or variable. "PutData" will send data specified within the command into a DADiSP variable or window. The "Execute" command will perform any DADiSP command as outlined by the help files. This is arguably one of the more important methods because it is extensible for future version of DADiSP. As new commands are implemented, the ActiveX API doesn't need to be extended, because the framework to

execute a new command or function is already there. In some cases, the "Execute" method may be used instead of "GetData" or "PutData". *("DSP Development Corporation: ActiveX")*

2.3 Component Object Model

Component Object Model (COM) is a conceptual and logical construct developed by Microsoft in 1993. ("Component Object Model") In a nutshell, According to Microsoft *"COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services.*" ("COM: Componet Object Model Technologies") COM is language agnostic, meaning that developers can take advantage of COM whether they write an application in Visual Basic, Java, C++, C# or even other proprietary languages. COM itself, is not a specific technology, but rather describes an evolutional series of technologies that include Object Linking and Embedding (OLE), COM+, Distributed COM (DCOM) and ActiveX. ("Component Object Model")

COM has been implemented on multiple platforms, however because it is a Microsoft product, it has the highest level of support and functionality on Microsoft Windows. Because COM was introduced in 1993, not all pieces of the suite were released out of the box, many were developed later. As later operating systems were released, such as Windows 2000, Windows XP and Windows Vista, support for newer technologies was included in the OS, while older operating systems required patches or updates in order to take advantage of the technologies. ("COM: Componet Object Model Technologies")

2.3.1 Object Linking and Embedding 1.0

Object Linking and Embedding (OLE) 1.0 is developed in 1990, prior to the term COM being defined. ("Object Linking and Embedding") However, OLE 1.0 is evolved out of Dynamic Data Exchange (DDE) that dates back to 1987 with Windows 2.0 and is still part of new windows releases, however is not utilized as often as newer technologies like OLE and ActiveX ("Dynamic Data Exchange"). Both DDE and OLE 1.0 are designed with the purpose of passing information between multiple documents, such as Microsoft Word or Excel ("Object Linking and Embedding")

DDE can transfer a limited amount of data between two documents, however OLE can create an active link between two documents or even embed one document within another as an object to transfer and reference data between the two without having to reconnect back to the document every time a change occurs like DDE does.

For Example, a Microsoft Excel document can be embedded into a Microsoft word document. When this happens with OLE 1.0, a visual representation of that document is displayed to the end user as a picture (such as a bitmap) and the data is stored in its own native format, i.e. excel format ("Object Linking and Embedding"). To show how this embedding works, below is an example using Microsoft Windows Vista and Microsoft Office 2007.

Figure 6 shows an example of OLE in action. A Microsoft Word 2007 document has an embedded excel spreadsheet within it. The excel spreadsheet looks and feels to the user just like a full version of excel, except it is part of the word document. When the user is editing the document, the toolbar ,redesigned and renamed to the Ribbon in Microsoft Office 2007, within Microsoft Word shows the formula bar. Also all of the toolbar tabs native to excel are merged together with the Microsoft

Word Ribbon. Looking at Figure 6 closer you can see that even multiple spreadsheets can be added and embedded within the same object.

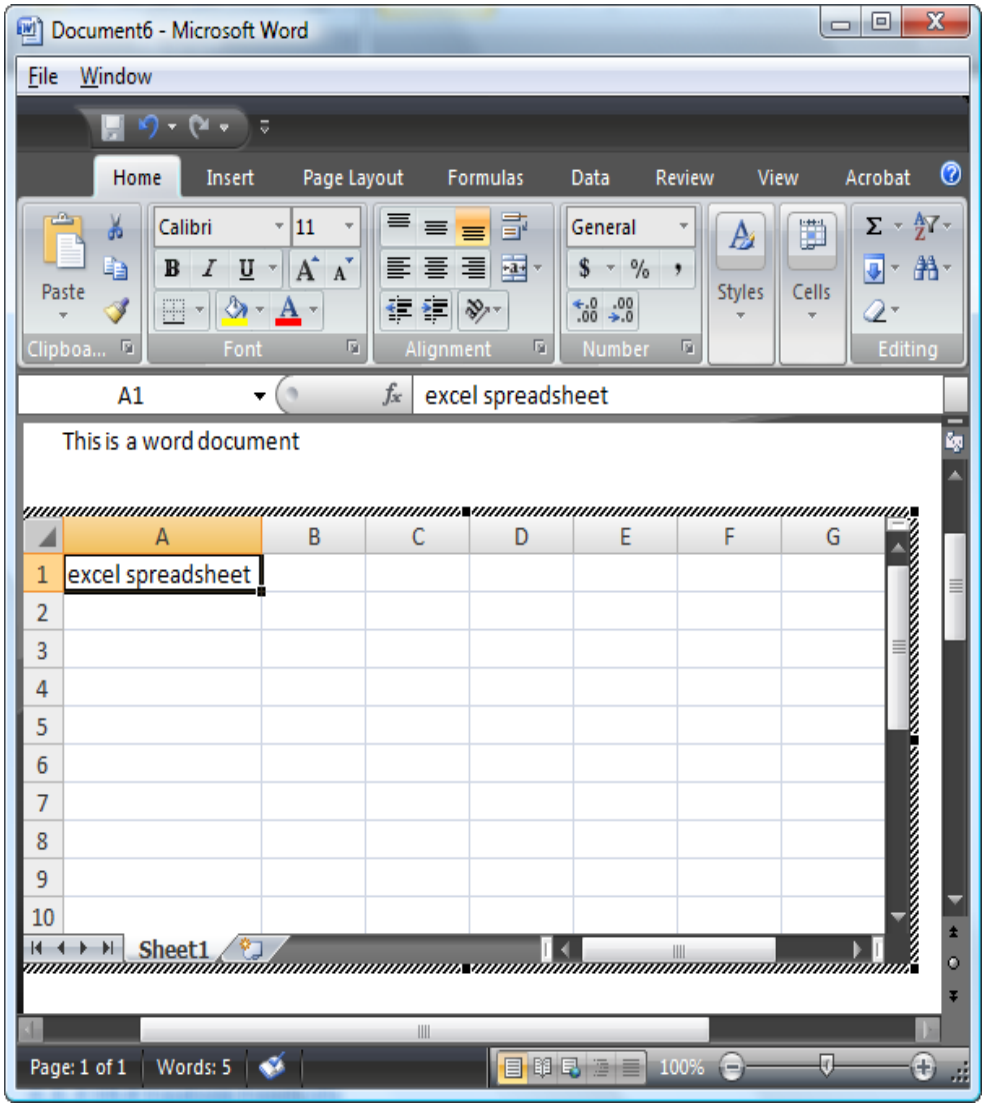


Figure 6 - Excel Worksheet Embedded in Word Document being edited

Taking a look at the Windows Task Manager in Figure 7 shows EXCEL.EXE is running under the processes tab but Microsoft Excel does not appear on the task bar. Microsoft Word has actually created and executed the EXCEL.EXE process so that it is not available directly to the end user, but must be utilized through the embedding or linking in Microsoft Word. In other words the base process is

executed, but the user does not have to wait for the entire application to load, yet is also still able to edit the object as if it were an entire Microsoft Excel Document.

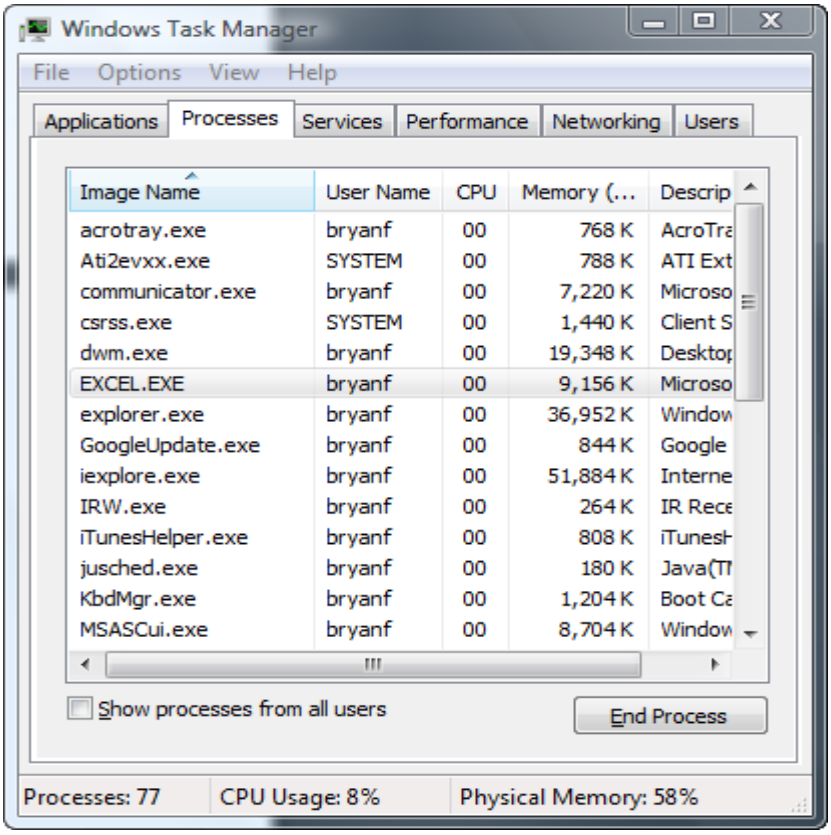


Figure 7 - Task Manager with Embedded Document

Once the user exits the spreadsheet but moving the cursor outside the bounds of the object, the object reverts to a simpler view that appears as just a table. The Ribbon has reverted to only the tools specific to Microsoft Word and the formula bar has gone away. However, taking a look at the task manager shows that EXCEL.EXE is still running. With DDE, once the user exits the embedded document, the process that allowed editing of that document exits. However with OLE 1.0, the process continues to be loaded into memory, but it is kept as a background process. If the user would like to make a change to the embedded document, it requires double clicking on the object and then the connection back to the process is moved the foreground. This is also faster because instead of recreating and executing the new EXCEL.EXE process, the existing one is just moved to the foreground.

Figure 8 shows the Microsoft Word document after the user clicks out of the embedded excel document and into the regular document. The excel piece is reverted to the simplified view.

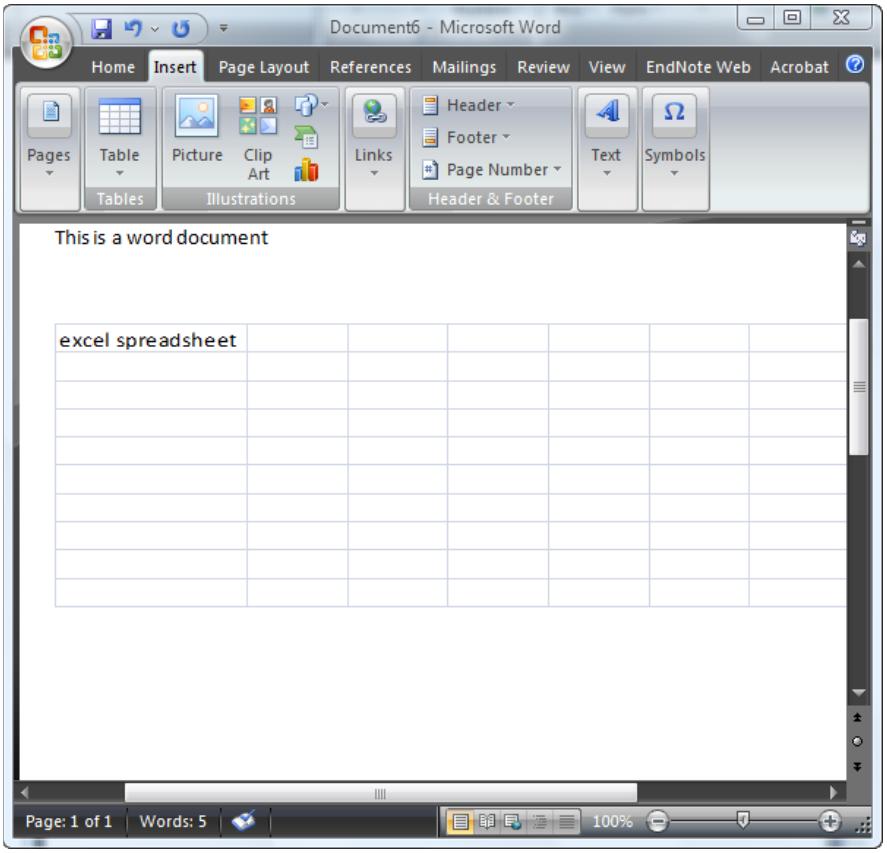


Figure 8 - Excel Worksheet Embedded in Word Document not being edited

2.3.2 OLE 2.0

OLE 2.0 accomplishes many of the same goals as OLE 1.0, however it was rewritten with the forethought of the COM framework. OLE 1.0 utilized system libraries and communicated to them via Virtual Functional Tables (VTBL), which are a list of pointers to various functions. The reimplementation of OLE 2.0 on top of COM allows it to be much more general and support new features such as

automation, drag-and-drop and structured storage. VTBLs are no longer used, however system libraries are still utilized. (“Object Linking and Embedding”)

OLE Automation is a standard defining how a program opens up its API so that another application, such as a visual basic script (.vbs) or even a Microsoft batch file (.bat), may automate the use of that program. This makes the script an ActiveX client and the application is an ActiveX server. This will be discussed in further detail in section 2.3.6 ActiveX. (“OLE Automation”)

OLE structured storage is used as a method for storing hierarchical data in a single file. Examples of this would be how a Microsoft SQL database that uses a single binary MDB file to store many tables, constraints, security information and user accounts. All the information is stored within the single file as well as a logical descriptor of how each piece of data relates to the others. Similarly Microsoft Office used this method for storing data in Microsoft Word and Excel documents, however Office 2007 is migrating to XML based Office Open XML (OOXML) format and supports binary OLE based files as well as files that is described by eXtensible Markup Language instead of OLE structured storage. OLE structured storage allows for application developers to utilize binary file types, without having to worry about an individual component of a file getting larger or smaller and then needing to determine what address in the file each component will now be stored in. (“Structured Storage”)

OLE drag-and-drop is almost exactly described by its name. An object, such as an icon, a file, or a control in a program can be clicked on (and hold the click) by the user with the mouse icon and dragged to another location. This is most common in Windows by moving icons around on the desktop. Figure 9 displays an icon being dragged from one location to another.

Figure 9 - Drag-and-Drop example

Meanwhile Figure 10 shows the icon after it has been dropped. These two figures utilize icons on a desktop within Windows Vista to show how drag-and-drop works. Many applications utilize drag and drop within them, including Microsoft Office and MathWorks SimuLink. In Microsoft office, text can be highlighted and then Dragged and dropped to another location within the document. In a SimuLink Model, one block can be dragged to another location or a new block can be dragged from the model library to a particular model. These are all examples of OLE drag-and-drop. ("Drag-and-drop")

Figure 10 - Icon Dropped

2.3.3 OLE Custom Controls

Visual Basic Extensions (VBX) were released as part of Visual Basic 1.0 to allow extensibility of the Visual Basic Language. An extension would be something like a radio button or toggle switch that could be a visual representation on a "form" that could then be linked to code in the background. ("Visual Basic Extension") In 1994, Microsoft deprecated VBX and instead promoted the use of more

versatile OLE Custom Controls (OCX). Usually OCX files are a dynamic link library (DLL) with specific definitions that are renamed to use .OCX as the extension. ("Object Linking and Embedding")

In 1996, Microsoft changed the standard definition of OCX so that the only one specific interface is required. This kept the file size down. Microsoft also renamed OLE Custom Controls to ActiveX Controls. ("Object Linking and Embedding") ActiveX Controls are commonly used within web browsers and for a time were a delivery method for spyware and adware. Familiar ActiveX Controls may be the Adobe Flash Player, Microsoft SilverLight or even the Java Runtime Environment. ("Component Object Model")

2.3.4 Distributed COM (DCOM)

Distributed COM (DCOM) extends COM objects so that they can be used on multiple computers. Until now, COM objects and controls have been confined to executing and existing on a single computer. DCOM allows for COM objects to be hosted on a single server computer via windows services and allow that service to be accessed over the LAN, WAN or Internet and provides tolerance for faults. ("DCOM Technical Overview")

DCOM allows the developer to not have to worry about the client side process crashing and leaving an open connection, or the connection between the two computers being lost and having a connection hung. DCOM deals with garbage collection and releasing resources when a client disconnects, gracefully or otherwise. DCOM also deals handles encapsulation of commands so that they can be transmitted over a TCP/IP connection that is serial, and then de-serializing the commands when they reach their destination.

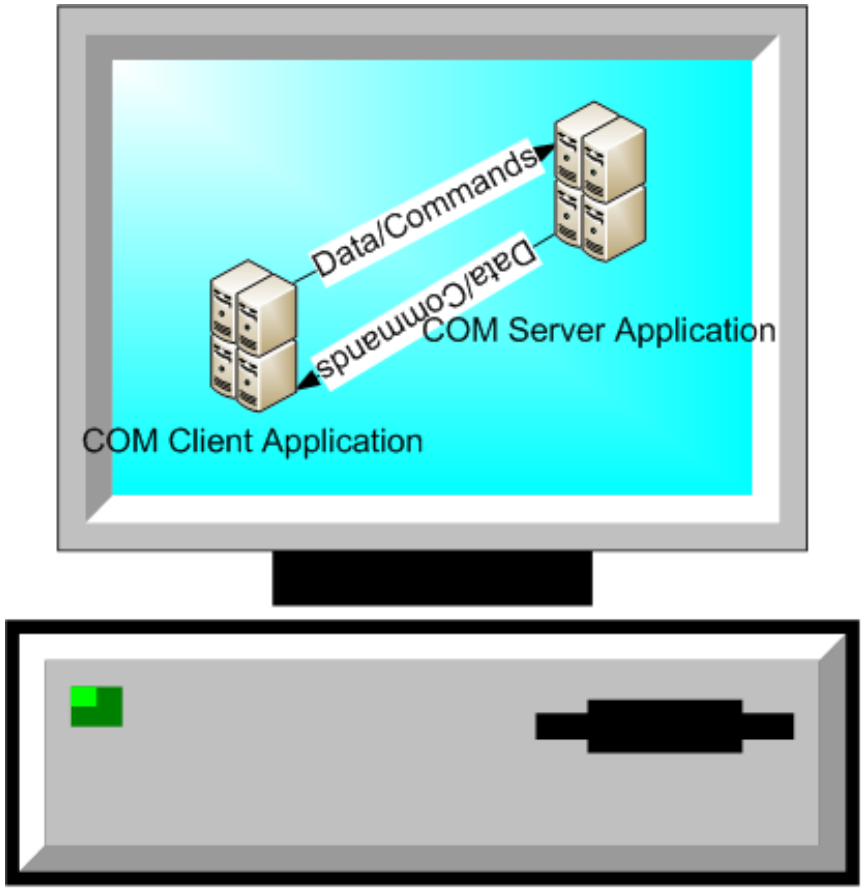


Figure 11 - Standard COM Application Interaction

Finally DCOM allows for a one-to-many relationship, where with standard COM there can only be a one-to-one relationship as shown in Figure 11. Standard COM applications have the client and the server application installed on the local system and all data and commands are localized within the computer. With DCOM applications, the server software is installed on a physical server computer. The client side application is installed on the client, or can be hosted on the same or separate server computer via a web site. Multiple clients can access the server application on the central server as shown in Figure 12 and Figure 13 respectively.

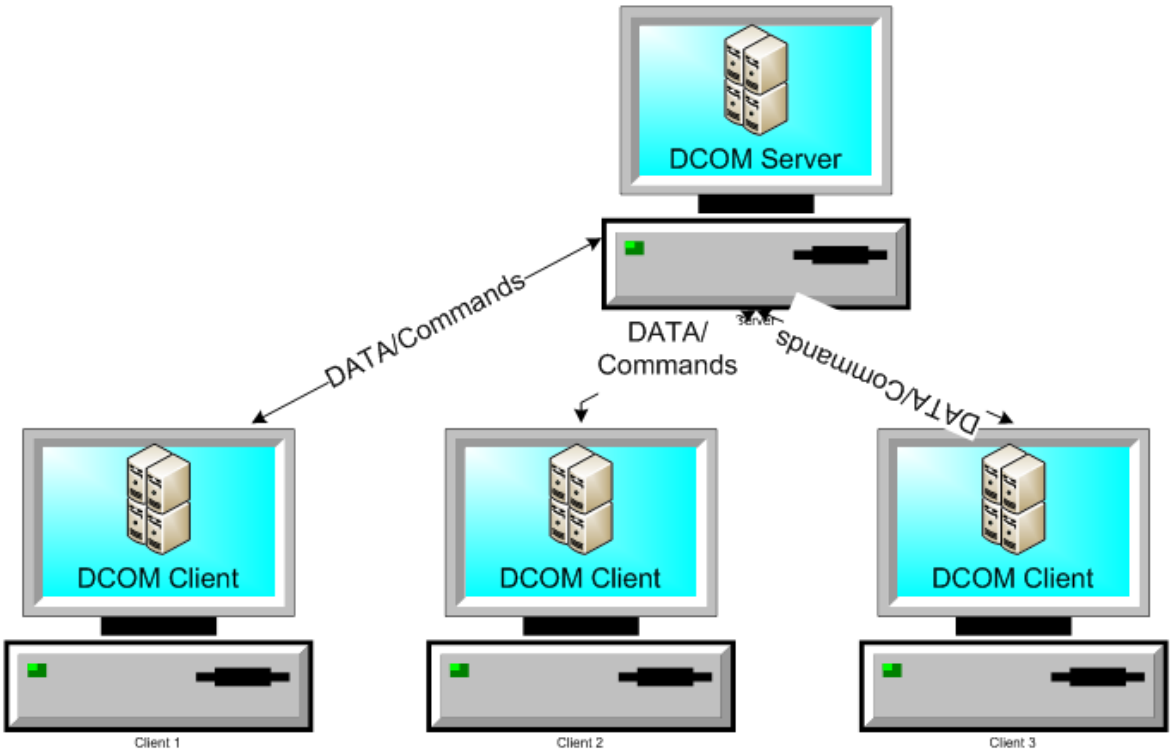


Figure 12 - Cleint/Server DCOM

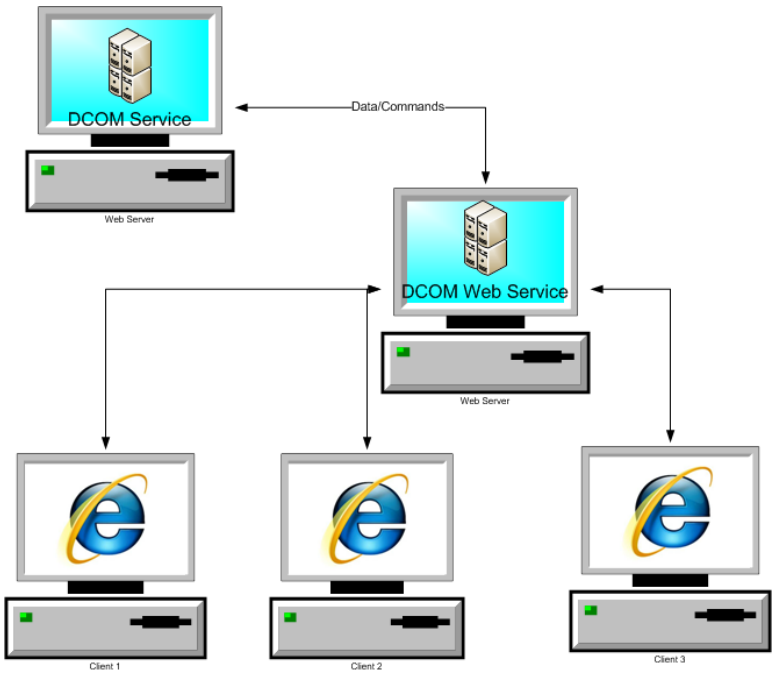


Figure 13 - Client/Web Server DCOM

2.3.5 COM+

In Windows NT Service Pack 4, Microsoft introduced Microsoft Transaction Server. This allowed developers a better way to deal with resource pooling, disconnected applications and distributed transactions, meanwhile offered better memory management and processor management. Microsoft Transaction server was an extension of COM and DCOM services. At the time Microsoft released windows 2000, they decided to incorporate this transaction server into the operating system and rename it to COM+. ("Component Object Model")

COM+ components were beneficial over standard COM and DCOM because they could be reused by new calls without reloading the components into memory because they were already there. Thus prevents multiple copies of a single routine from existing in memory. Components could also be distributed like DCOM, thus reducing the need for strictly DCOM components. ("Component Object Model")

2.3.6 ActiveX Automation

ActiveX Automation is evolved out of OLE Automation. Developers can open up specific functionality within their applications to provide an API to end users and other developers through standard ActiveX COM runtime. Applications that allow for ActiveX Automation enable programmers and users to script and control their application through external code. Many of Microsoft's own application are enabled for ActiveX Automation, such as Internet Explorer and Microsoft Office and even Active Directory. ("ActiveX")

Microsoft Excel, for example, can be manipulated by an end user through a Visual Basic Script that connects to excel, opens a file performs particular formatting on a file and then saves the file, even with a new name to preserve the old content. This automation task would be useful to someone

working with a large number of reports that are output from an application to an excel format that is not readable right away. The visual basic script can automate the process of manipulating the reports so that they are all uniform and visually appealing and save significant time and tedious labor.

Another Example of ActiveX Automation is a systems administrator creating user accounts in an Active Directory Domain. A systems administrator my receive a list of new employees to start at a company. The administrator may use a visual basic script to make a connection to Active directory. Once connected they can create user accounts, set a password, set an email address and even execute a remote Microsoft Powershell script that will create a Microsoft Exchange 2007 mailbox and link it to the appropriate user account.

2.3.7 Microsoft .NET

The Microsoft .NET Framework is standard set of tools, utilities and solutions to common programming problems to allow more complex solutions to more complex problems to be solved more timely. Microsoft .NET also incorporates a runtime environment of its own so that code is more portable. It also allows developers to develop uniformly between desktop and web applications and use the same code in both places. This allows users to have a seamless experience on the desktop as well as on a web client without significant extra effort from the developers. ("ActiveX")

> *"The .NET Common Language Runtime provides bi-directional, transparent integration with COM. This means that COM and .NET applications and components can use functionality from each system. This protects your existing investments in COM applications while allowing you to take advantage of .NET at a controlled pace. COM and .NET can achieve similar results. The .NET Framework provides developers with a significant number of benefits including a more robust, evidence-based security model, automatic memory management and native Web services support. For new development, Microsoft recommends .NET as a preferred technology because of its powerful managed runtime environment and services." ("COM: Componet Object Model Technologies")*

.NET has gone through many revisions and the currently released version is 3.5 Service Pack 1 and still supports COM. Microsoft has not as of yet deprecated COM in favor of .NET, but rather used the two in conjunction to enhance one another. There is no set date to deprecate COM in the near

future, even though some COM functions may seem redundant as they have been implemented within .NET. (“ActiveX”)

Microsoft .NET Framework is fully supported on Windows XP, Vista, Server 2003 and Server 2008 and also has a compact framework for Windows Mobile smartphones to allow seamless applications across the Desktop, Web and now cell phones. (“ActiveX”) An open source project known as mono is dedicated to providing an open source, cross platform development and runtime environment for .NET. Meaning that .NET applications, if developed correctly, could even run on Linux, Unix and Mac OS. (“Mono”)

3. Development & Testing

This section will discuss using the development and testing that occurred in creating a connection between MATLAB, SIMULINK and DADiSP as well as passing data and commands between the applications. Much of this discussion focuses on sending commands and data to DADiSP and learning the DADiSP language as well as how to construct a Level-1 S-Function for use within Simulink. Some of the more difficult design problems focus on working within the SIMULINK environment which is a runtime environment and has a somewhat cyclical nature. This can make coding and troubleshooting difficult. For the purpose of Development and testing, I used MATLAB 2008a, Simulink 7.1 and DADiSP 6.0 provided by WPI's academic licenses.

3.1 Brokering Connection

Before any data or commands can be transferred between applications, some sort of link between the two must exist. This demo was built in steps starting by first making a connection from the MATLAB command window to DADiSP and passing data and commands from there. The next step was to move the compiled commands into an M-File that could be run from the command window. The Final step and largest connection step was to convert the M-File into a level-1 S-function and implement it within a SIMULINK model.

3.1.1 Passing Commands from MATLAB to DADiSP

The initial development began at the MATLAB command window. MATLAB, as well as most other ActiveX clients talk to the ActiveX server over a local variable within the application. To do this, we set a variable equal to the ActiveX server command that connects to a particular application via it's program ID. This command is "ACTXSERVER" in MATLAB and the program ID of DADiSP is "DADiSP.Application". So to create the connection, we used the following command.

```
>> hDADiSP = actxserver ('DADiSP.Application');
```

Using the “ACTXSERVER” command in this way connects to the ActiveX Server on the local computer. It is, however, robust enough to connect to an ActiveX server on a remote computer by simply specifying the hostname of the computer as below. In this case, of course there is additional complexity of permissions and firewalls so it is only noted in here.

```
>> hDADiSP = actxserver ('DADiSP.Application','Computername');
```

Looking at the current workspace shows a new variable “hdadisp” that looks just as if it were a Matrix of statistical or signals data. This variable is the key that can send commands and transmit data back and forth with the DADiSP application. All ActiveX methods exposed from DADiSP are available on this variable. But once this variable was set, the DADiSP window did not appear, however the task manager now showed DADISPNT.EXE running when it was not prior. So now DADiSP just needed to be made visible. To do this we executed the following.

```
>> hDADiSP.visible = true;
```

This exposed the DADiSP Command window as shown in Figure 14 with a blank worksheet with default 4 blank windows. At this point DADiSP is both controllable by the user through its own graphical user interface, or MATLAB can also send commands to it from the command window, however the last change that is executed will be overwritten.

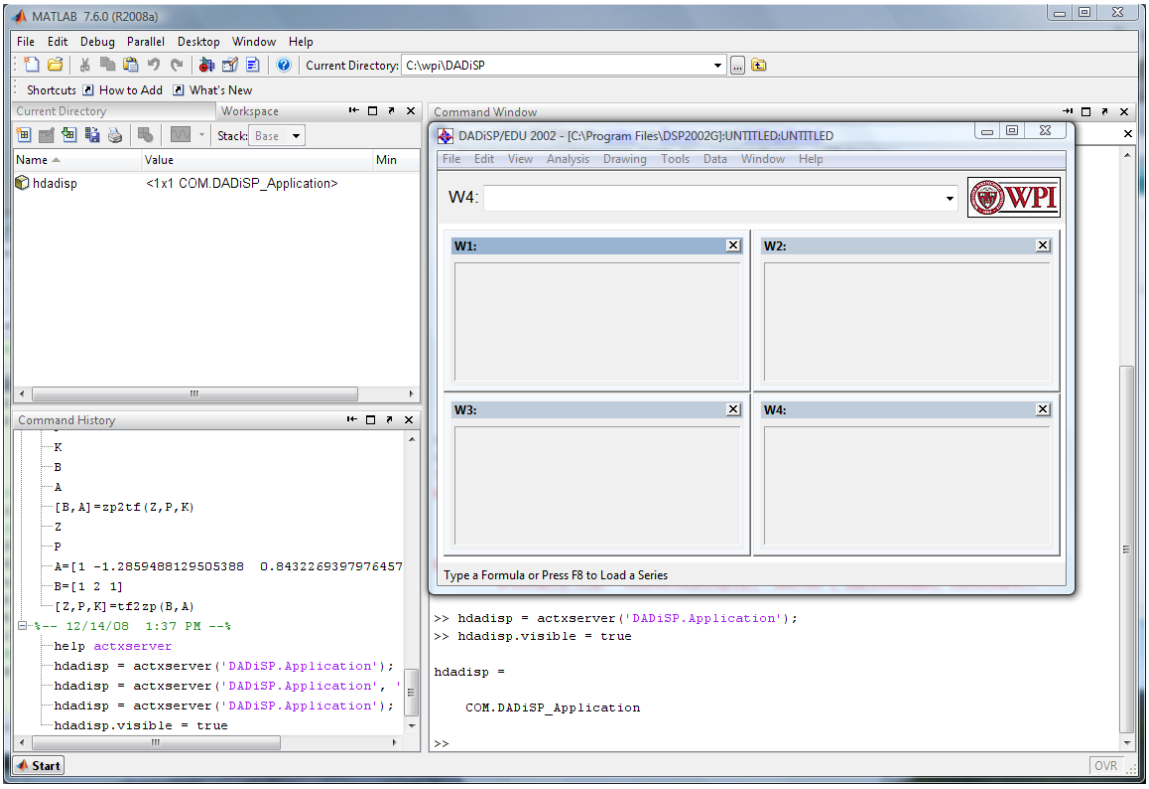


Figure 14 - DADiSP Opened and Visible from Matlab

Now that DADiSP was open, we could visually see some of the formatting we might want to make using the "EXECUTE" method. The "EXECUTE" method is not restricted to making formatting changes, but it is the only way to do such changes. Also these changes can be made while DADiSP is not visible, they will be shown once DADiSP is made visible again. The following commands perform some of these formatting changes and are shown in Figure 15.

```
%Create a new blank worksheet with 4 windows
>>hdadisp.Execute('NewWorksheet(4, 0)');

%Hide the DADiSP toolbar
>> hdadisp.Execute('setconf("TOOLBAR_ENABLED","0")');

%Set the X-Axis of Window 1 to be "t (sec)"
>> hdadisp.Execute('SETXLABEL(W1,"t (sec)")');

%Change the Lable of Windows 1 to be "Matlab Data"
>> hdadisp.Execute('label(W1, "MATLAB Data")');
```

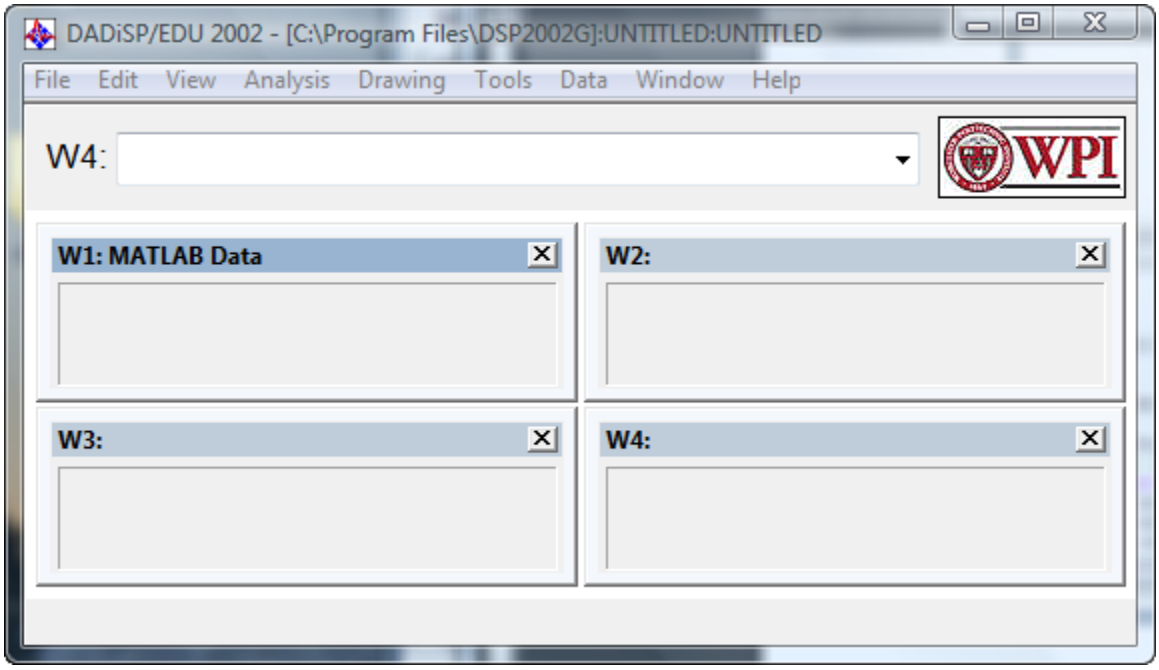


Figure 15 - DADiSP after formatting Mods

3.1.2 Bidirectional Pass of Data & Controls

In order to transfer Data to DADiSP, we first needed some sample data. WAV files are readily available to anyone with Microsoft Windows as they come preloaded for notification purposes such as application errors and pop ups. WAV files are also easily converted into a Matrix in MATLAB as they are already raw audio files. The files provided with Microsoft Windows are single channel audio files, or monophonic as opposed to stereo files that will have two different signals to worry about. This sample file was also useful later for the SIMULINK model.

To convert this file into a MATLAB variable, we used the "WAVREAD" command. "WAVREAD" will also determine the sampling frequency and number of bits used to encode each sample. The sampling frequency would be useful later for determining the correct axis on a graph be it within MATLAB or DADiSP.

```
>> [data,fs,nbits] = wavread('sample.wav');
```

So to send data to DADiSP we used the "PUTDATA" method. All we need to specify with this method is the Window we are sending the data to and the variable containing the data. In our case we actually used the variable "data" which contains the WAV file we imported earlier. The command looks like the following and the DADiSP window now appears as in Figure 16.

```
>> hdadisp.PutData('W1',data);
```

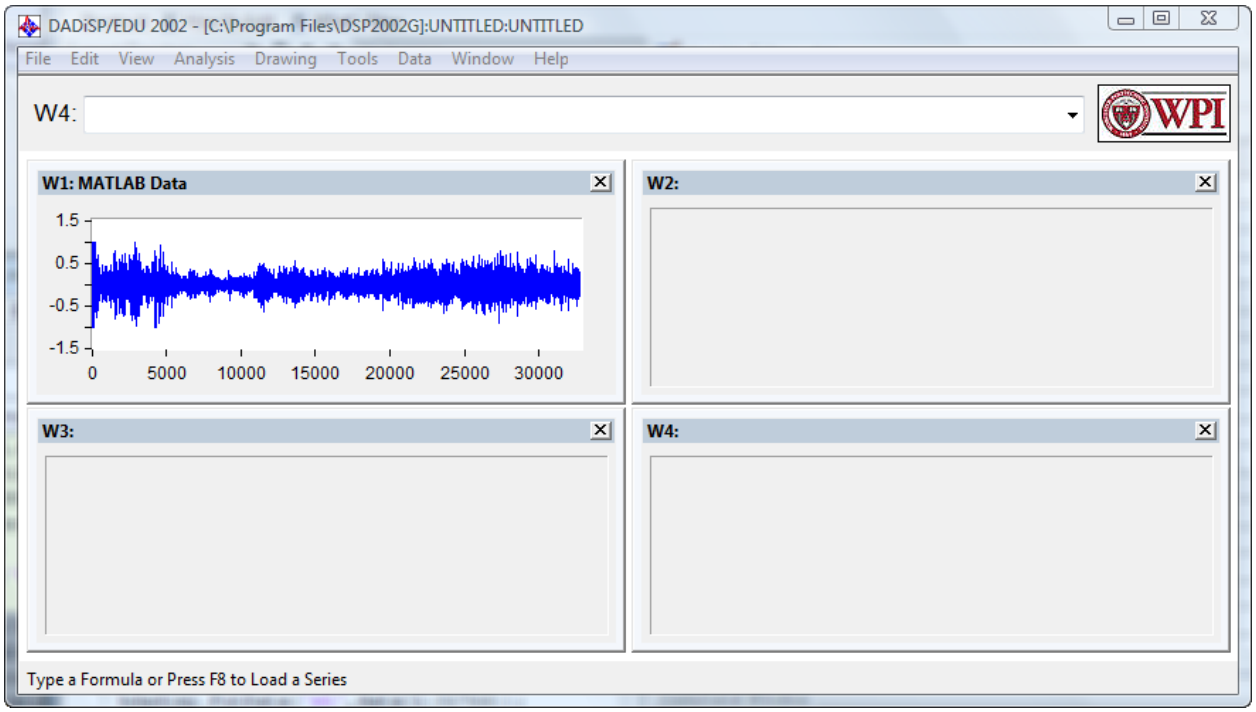


Figure 16 - DADiSP window with data from MatLAB

This looks visually like one would expect, however the Axis are incorrect and it's just raw data, so we need to make it look a little better. We know our sampling frequency before and we can also figure out how many samples are shown. To perform more formatting we will need to use the "EXECUTE" method again. However the "EXECUTE" method only accepts a string for input and some of the changes we need to make are variables within MATLAB. To overcome this challenge we will need to concatenate a string together and pass that string into the "EXECUTE" method as one command. The resulting commands look as follows and the DADiSP Window looks like Figure 17.

```
%The total Time of the file is the number of samples in the file divided by the
sampling frequency.
>> execute_time = t_max/fs;

%That needs to be converted to a string
>> execute_time=num2str(execute_time);

%then concatenate the string together with the command to set the distance
between points on the X-Axis
>> deltax_cmd = strcat('SETDELTAX(W1,',execute_time,')');

%Execute the command
>> hdadisp.Execute(deltax_cmd);

%Set the Label on the X-Axis (overwrite if already done)
>> hdadisp.Execute('SETXLABEL(W1,"t (sec)")');
```

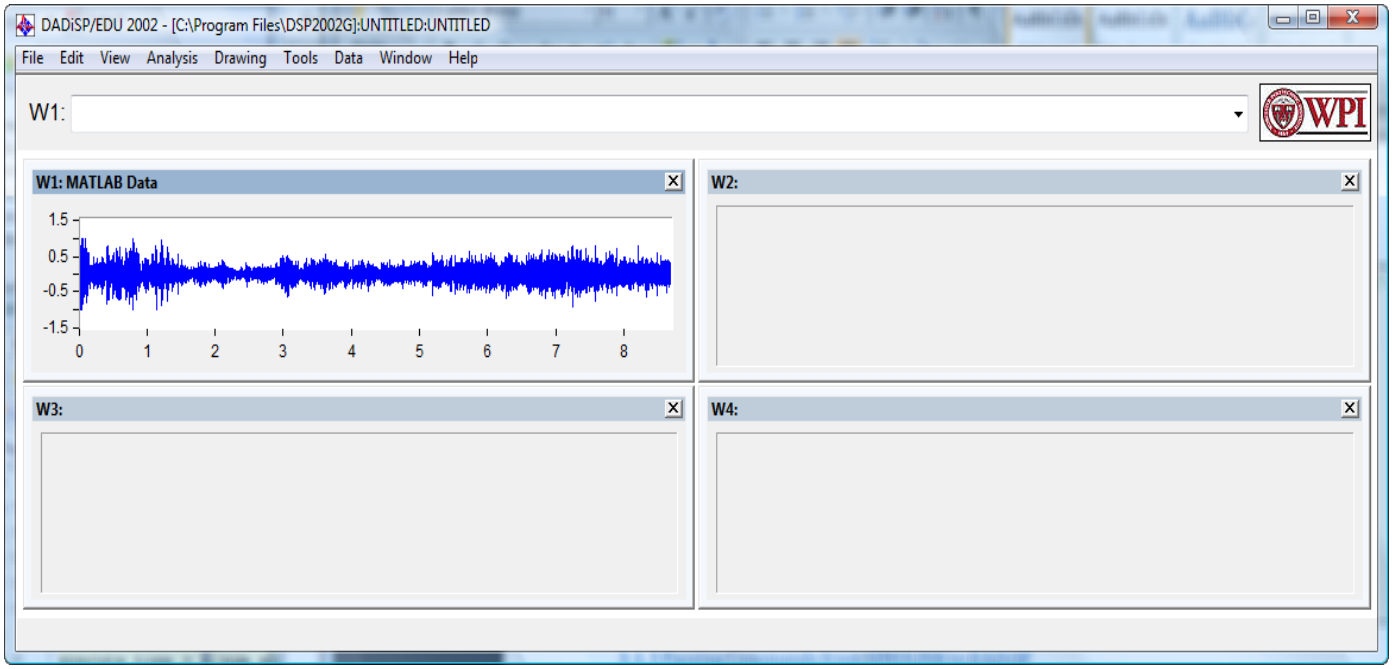


Figure 17 - DADiSP with Formatted MATLAB Data

After we got the data formatted properly in DADiSP, we could start using the tools within DADiSP for manipulation. We did this, however through the ActiveX connection from MATLAB, not through the DADiSP interface itself. For example, we put the Spectrum of Window 1 into Window 2 and labeled it using the following code and shows in Figure 18.

```
%Put the FFT in Window 2
>> hdadisp.Execute('W2=Spectrum(W1);');

%Label Window 2
>> hdadisp.Execute('label(W2, "Spectrum of Window 1")');
```

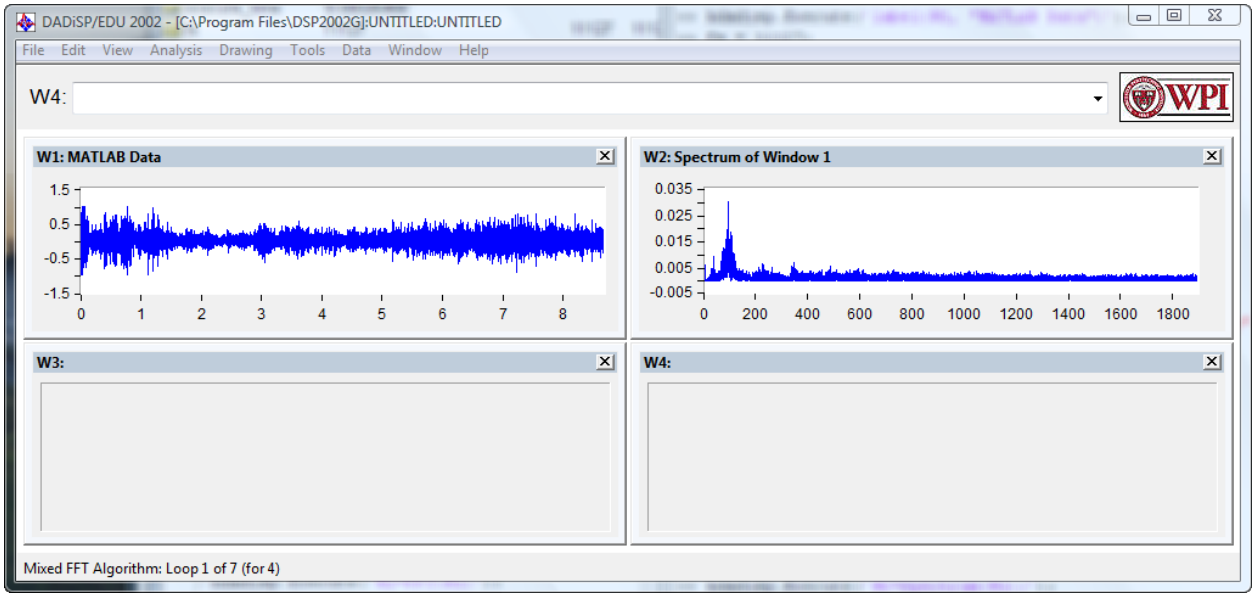


Figure 18 - Spectrum and labeling of Window 2

Various other commands can be executed in this way including everything that is listed in the "DADiSP Functions" help menu. Anything that can be executed as part of SPL can be executed via ActiveX through hdadisp.Execute. From here, the much more difficult task is to integrate this now with SIMULINK, a continuous real-time environment.

3.1.3 Creating the SIMULINK to DADiSP Connection

Creating the Connection between SIMULINK and DADiSP is a much more difficult task than creating a connection from MATLAB. The reason is that SIMULINK is a continuous runtime application. A custom, user defined function within MATLAB is a specially defined M-File called an S-Function. The way the S-function works, is it takes in the first data point from the source then runs executes

everything within the output section of the S-Function. For the next data point it repeats this process all over again.

This poses a large challenge when needing to instantiate a connection to a server or application because each time the S-File executes it will try and create the connection all over again. Unfortunately it's even worse when connecting to an ActiveX server because it will create a new process for the remote application each time it cycles through the S-Function. If your data series has 30,000 data points, then it will try to create 30,000 DADiSPNT.EXE processes, or one for each data point. This would cause the host system to run out of memory and crash very quickly and is also useless for data processing because each data point will be spread out across 30,000 different sessions and cannot be analyzed.

To get around this we had to devise some sort of global variable or preference that was external of the S-function and could be checked to see if the DADiSP connection was already created. The Windows Registry Database is the logical choice for this to be created. The windows registry is a hierarchical database used to store preferences for programs and windows settings themselves. There are different keys for the local system (global) as well as for the current user that is running. In order to have this application usable by non-administrators, we needed to put this in the current user registry Hive, or database file, named HKEY_Current_User.

```
>> dadisp_status=winqueryreg('HKEY_CURRENT_USER', 'Software\WPI\DADiSP', 'Turned_On');
```

The above command using "winqueryreg" will query a particular registry value and stores the value into the variable "dadisp_status". In order for this to work, the 'HKEY_Current_User\Software\WPI\DADiSP\Turned_On' value must exist. That part is taken care of during the startup scripts that will be talked about further later on. The value of "Turned_On" will be a

string character. We used a bit more logic to test the value and if it is set to '0', then it will create the variable that initiates an ActiveX connection to the DADiSP application and then it will overwrite the value of "Turned_On" to equal '1' to avoid creating another connection to DADiSP next time. If the first read of "Turned_On" is already non-zero, then it will assume that the connection has already been made and continue to process the remainder of the script. The following block of code illustrates this mechanism.

```
   %Logical Check agains the DADiSP Status from Registry.
>> if (dadisp_status == '0' )

   %Tell the registry that DADiSP is on
>> regFileName = 'c:\temp\DADiSP_On.reg';
>> fp = fopen(regFileName,'wt');
>> fprintf(fp,'REGEDIT4\n');
>> fprintf(fp,'%s\n','[HKEY_CURRENT_USER\Software\WPI\DADiSP\]');
>> fprintf(fp,'%s%s%s%s%s%s\n','"', 'Turned_On', '"=', '"1"' );
>> fclose(fp);
>> dos('%windir%\regedit.exe /s c:\temp\DADiSP_On.reg');

   %Create activeX connection to DADiSP ActiveX server
>> hdadisp = actxserver ('DADiSP.Application');
>> hdadisp.Execute('NewWorksheet(4, 0)');
>> hdadisp.Execute('setconf("TOOLBAR_ENABLED","0")');
>> set (hdadisp, 'Visible', 1);
>> else
  %Do nothing because DADiSP is already active.
>> end;
```

An important piece to note is that MATLAB has a function to read directly from the registry, but does not have a function to write directly to it. In order to accomplish this task we created a method that writes a .reg file to the c:\temp directory and writes out the appropriate registry information into a format that is understood by the windows Registry application (regedit.exe). Then we used a windows command prompt with the "dos" command to execute and install the registry file. After that we make a connection to DADiSP, setup the worksheet and make it visible. Creating this scheme for checking if DADiSP is already running adds minimal load to the application because it is only going to actually

execute tasks and Write the .REG file out to disk the one time that it needs to execute, rather than every time.

There was one more problem, however that is significantly more difficult to deal with. When connecting to DADiSP through the MATLAB command window or an M-File, everything was done sequentially and asynchronously. It didn't matter when the WAV file was converted into a matrix, as long as it was converted prior to using the "PutData" method to put data into a window. Also the whole WAV file was converted and then all at once pushed up to DADiSP "PutData". In the SIMULINK world, the whole data file doesn't exist yet because each sample is being processed one at a time, just like it would be in real time.

The "PutData" method would not work in this case because the data only exists one point at a time, unless it could somehow be stored in a buffer. But using a buffer results in a delay between input and output, which for some use cases, is valid. However, For a single set of raw data of duration 10 seconds, would typically require more memory than a 32-bit windows computer can address, which is 4,096 MB maximum. The "PutData" method could be used if the full contents of the DADiSP window were read back into a variable, add the one new value, then write the entire string back to DADiSP as shown below.

```
% Read Back the Data from the DADiSP window and put it into the
variable "temp"
temp = hDADiSP.GetData('W1');

%Add the current input value, 'u', to the end of the array "temp"
temp = [temp;u];

%Write the new 'temp' variable back up to DADiSP
hDADiSP.PutData('W1', temp);
```

This resulted in additional memory issues as the "temp" variable became again, as large as the data file and every round, MATLAB was looking for a new block of memory that would fit the previous

variable plus the concatenated value. Creating a variable of appropriate size at the beginning was fruitless because the memory required is large and in the end depends on the file size of the input file. This was not a reasonable approach.

To solve this issue, I had to utilize a DADiSP command called “Curr”, which is used as the current contents of the window. This command is only available to ActiveX clients through the “EXECUTE” method so using it with variable input required converting numbers to strings and concatenating commands again. Also, in order to use the “Curr” command, we need to, within DADiSP, concatenate the command itself with the next value to be added. The result of these stringing concatenations together looks like the following.

```
   %Focus on Window 1
>> hDADiSP.Execute('Moveto(W1)');

   %Convert the input ‘u’ from a number to a string for
   %concatenation.
>> v = num2str(u);

   %Concatenate the DADiSP concatenation command with the values
   %that need to be concatenated.
>> newdata=strcat('CONCAT(curr,{',v,'})');

   %Execute the command
>> hdadisp.Execute(newdata);
```

Lastly, there was one challenge to face: how to get the output from DADiSP individually bit by bit. The length of the output isn’t necessarily known at any given time to use as an index. It could be kept track of with a counter variable, but that means more wasted memory and isn’t necessarily going to be correct if something changes the window externally. To cope with this, I started by creating an extra DADiSP window used for calculations and hide it using the “hide” command to pass back special information.

```
   %Hide Windows 6
>> hDADiSP.Execute('hide(W6');
```

With this hidden window, I could now grab the length of the particular dataset in the window we want to grab and throw it into the hidden window. We can import that information into a simulink variable, since it's only one value it will not be a big variable and time creating the variable will be minimal. Then we need to convert that number to a string in order to concatenate it with the command to extract just the value of the output window at that particular index we want and put it into the hidden window. Then we pull the data out of that window and output it out of the S-function as shown below. They output of the S-function is, by default, the "sys" variable. The "sys" variable is defined below as the data from

```
   %Put the Length of the window you want to output from into a
   %hidden window
>> hDADiSP.Execute('W6=COLLEN(W5,1)');

   %Get the length into a MATLAB variable.
>> lengthW5=hDADiSP.GetData('W6');

   %Convert Length of Window 5 to a string.
>> slengthW5 = num2str(lengthW5);

   %Make the Command String to Get the Last Value of Window 5 (output window).
>> cmdgetoutput=strcat('W6=W5[', slengthW5, ']');

   %Execute the Command to put the Last value of Window 5 in Window 6.
>> hDADiSP.Execute(cmdgetoutput);

   %Output the contents of Window 6 out of the S-Function as the result.
>> sys = hDADiSP.GetData('W6');
```

Combining all of this code together into a single S-funtion provides a single connection to DADiSP, sending data to DADiSP and retrieving data from DADiSP in effective ways that don't drastically increase the resource consumption of the host computer. From here it was a matter of creating a sample model that utilizes this S-function and provides input from a binary audio file.

3.1.4 Creating Sample Model

To create the Sample Model, The first piece was getting data. I used the same sample.wav file that I used when discussing initially how to transfer data to DADiSP. Simulink uses has a block that performs an analogous function to “wavread” called “From Wave File” as shown in Figure 14. With this block, simply specify the name of the file, within the same directory, that needs to be imported and it automatically figures out the sampling frequency, number of channels and number of bits used to encode each value.

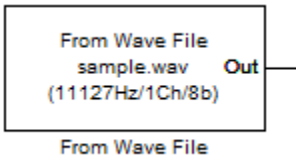


Figure 19 - From Wave File Block

After that we sent the data into an unbuffer to clean the data make sure it isn’t buffered as in Figure 20.

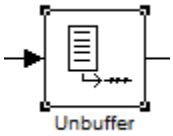


Figure 20 - SIMULINK Unbuffer block

The Data point then flows into the S-Function block where the major work happens.

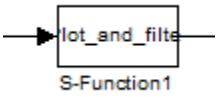


Figure 21 - S-Function Block

However the S-Function block is uninteresting without knowing seeing the parameters. The parameters of an s-function block are shown in Figure 22.

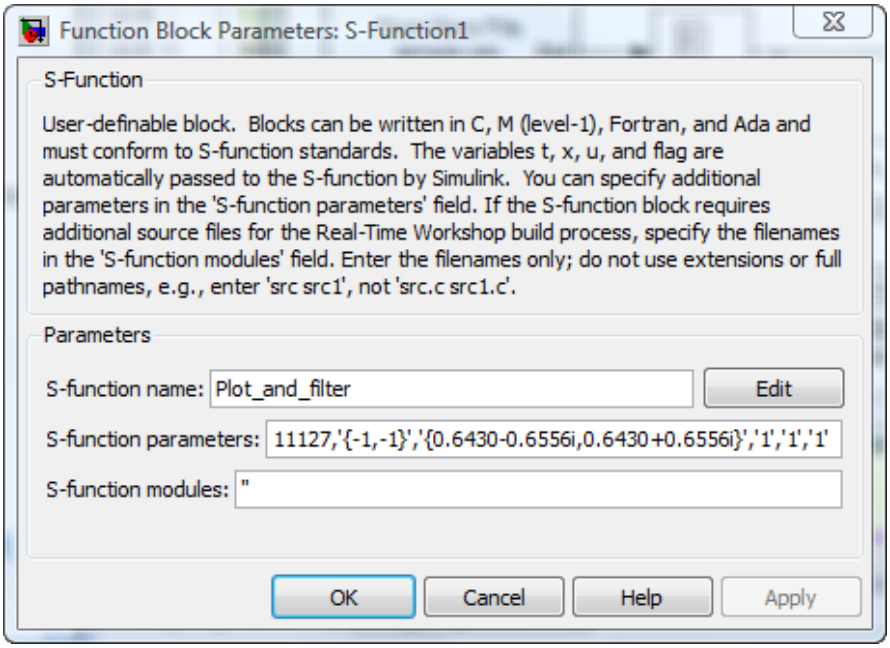


Figure 22 - S-function Parameters

The S-function name is the name of the custom M-File. The other S-function parameters are separated by commas. These are custom to the S-function itself and their variable types are specified in the M-File. The parameters are as follows

1. Signal Frequency
2. Array Zeros for filter transfer function specification
3. Array of Poles for filter transfer function specification
4. Gain for filter transfer function specification
5. Sample Rate for filter transfer function specification
6. Warping Frequency for filter transfer function specification (for bilinear transforms, not used in this example)

All of these pieces fit together in the final model as seen in Figure 23. Once these pieces were in place we could begin working on make DADiSP perform some more advanced functions on the input WAV to perform transformations or gain.

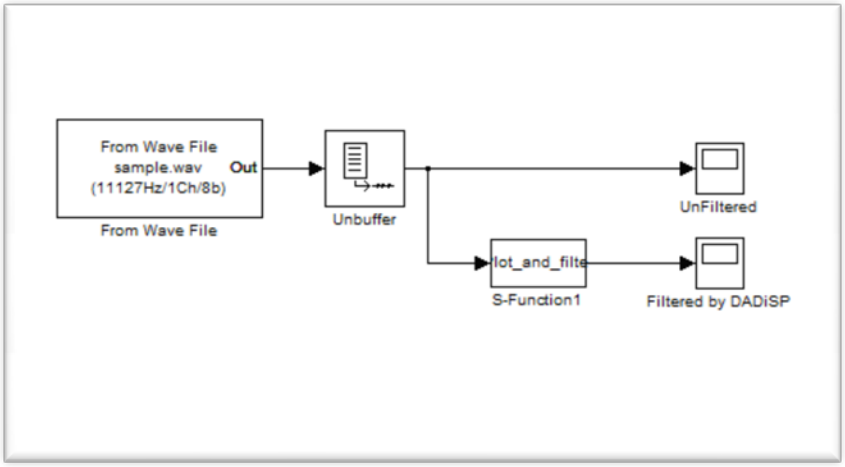


Figure 23 - Complete Model

3.2 Filtering

A Digital Filter is defined as "An algorithm operating upon a sequence of discrete-time sampled data, designed to pass signals with selected temporal or spatial frequencies while attenuating signals with other temporal or spatial frequencies." *("AMS Glossary")* Highpass filters will pass the high band while attenuating those below a specific breakpoint frequency. Lowpass filters perform the exact opposite task. A Bandpass filter will attenuate all frequencies except those within a specific range. *("AMS Glossary")*

Filters are typically described by their transfer function. In the analog world, H(s) or in the discrete universe, H(z), where H(z) is the ratio of the desired output response to the input. Where the output response and transfer function are described below. *("Z-transform")*

$$Y(z) = H(z)X(z)$$

$$H(z) = \frac{Y(z)}{X(z)}$$

The numerator has M roots, or Zeros and the denominator has N roots, or poles but the number of roots and poles can be the same. In addition, there may be a zeros and poles at z = 0 and z = ∞. The number of zeros and poles are always equal. *("Z-transform")* Using zeros and poles in conjunction with determining what frequency band is desired to be filtered, we can design filters using software such as dadisp and MatLab to filter digital signals.

3.2.1 Creating a filter in DADiSP

DADiSP offers a few different methods for creating a filter. Different methods are better for different circumstances. Also different methods may employ different mathematical functions in order to evaluate the results. These methods are described in the DADiSP help files.

Probably the simplest filters to implement are the Lowpass and Highpass filters. These are available through special functions within DADiSP. Their simplest form is a single pole filter. To utilize this function all that's needed is an input signal (or window) and a cutoff frequency. The command is "slp(<window>,<frequency>)". This command can be used from a DADiSP SPL file, an ActiveX client through the "Execute" method, or through the DADiSP GUI from the Analysis menu. A highpass filter can be created similarly by using "shp" instead of "slp".

Other More complex filters can also be generated in DADiSP using the Filter menu. Figure 24 shows a number of different filter options available to be designed.

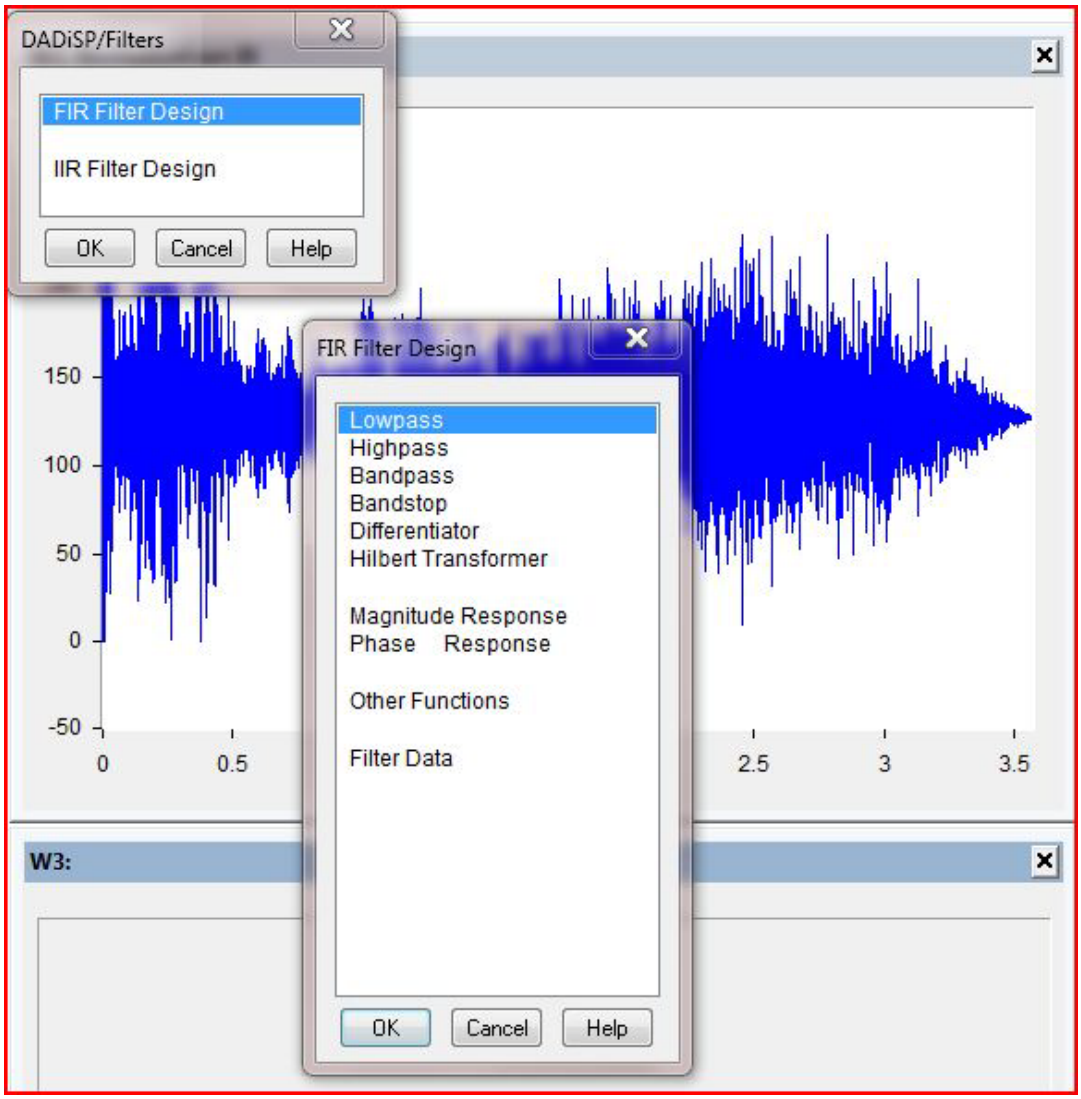


Figure 24 - Other DADiSP Filter Options

Lastly DADiSP can also create very granular filters using the Z Pole-Zero Plot. The menu shown in Figure 25 allows users to enter in the desired Zeros, Poles and Gain for a filter. The fields will accept real and imaginary numbers. All zeros and poles must be encapsulated in brackets and separated by commas. DADiSP also has an associated command line API for this tool. The command is zplane and it accepts the same arguments as described in the menu below and in the very same format.

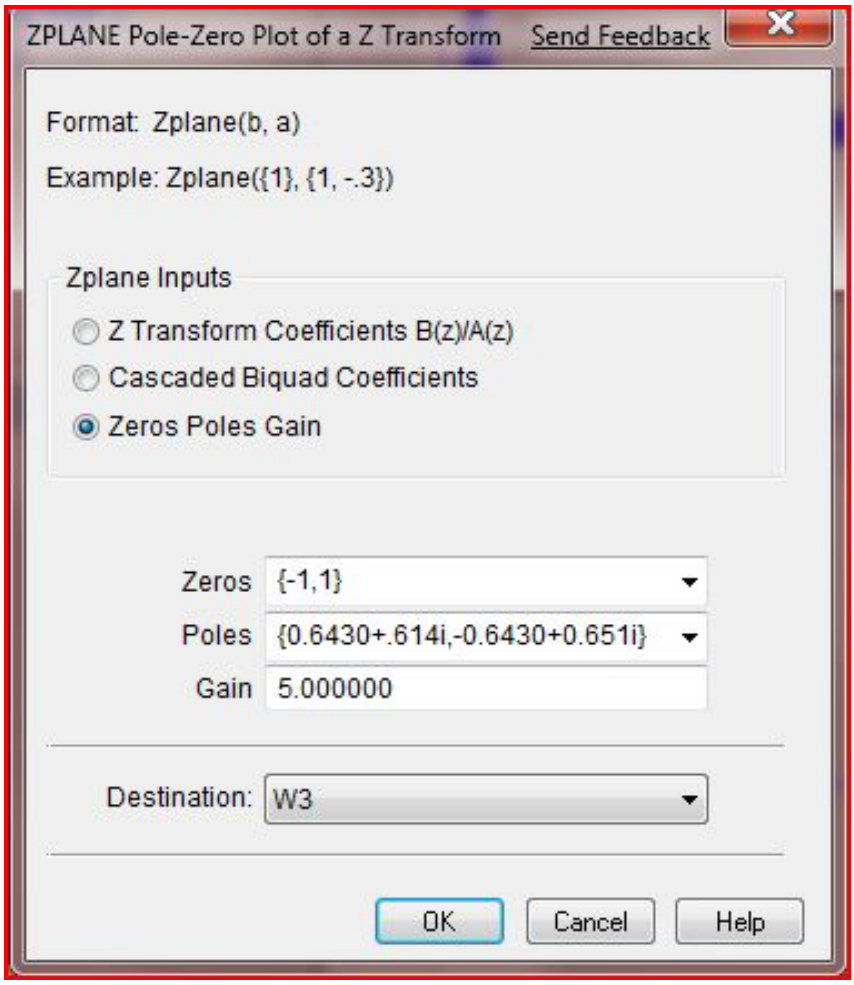


Figure 25 - Z Pole-Zero Plot Menu

This function draws a zero-pole plot of the zeros and poles in its own window. The user can the use the "cascade" command in a third window to overlay the input onto the filter plot window and receive the filtered output. If Window 1 contains the raw signal and Window 2 contains the zero-pole plot, then the filtered Window3 would contain the following command: "cascade(w1,w2)".

Both of the above mentioned filter methods within DADiSP have a command line version to execute them, they can all be executed from MATLAB and SIMULINK to DADiS over ActiveX. In addition, the filter and cascade commands only needs to be executed once because DADiSP will automatically update the output of a given window whenever the input is modified.

3.3 Comparing FFT Output

In order to verify that DADiSP and Matlab are performing equivelant tasks. In order to compare these two, I took a simple task, the Fast Fourier Transform (FFT) and compared the MATLAB calculated FFT with the DADiSP calculated FFT from the same input. The actual calculations were performed with a simple m-file script. To perform the comparison, however, the outputs are copied into excel and each real and imaginary value are compared individually.

The script used to calculate the two FFTs is shown below.

```
%This M-File is designed to input a signal, calculate the FFT of the
%signal.  Then it will also submit the signal to DADiSP the exact same
%process and will allow comparison of the output values.

input=wavread('sample.wav');

var_fft = fft(input);

hdadisp=actxserver('DADiSP.Application');

hdadisp.visible=1;

hdadisp.PutData('W1',input);

hdadisp.Execute('W2=fft(W1)');

var_dadisp_fft = hdadisp.GetData('W2');
```

DADiSP seperates the real and imaginary parts of the FFT into separate arrays, while the MATLAB output keeps them in the same array. Figure 26 shows the first 12 rows of the FFT output calculated by DADiSP for a two-channel signal. The real and imaginary parts of each channel are broken into their own column. While Figure 27 shows the first 12 rows of the FFT output calculated by DADiSP for the same two-channel signal. The real and imaginary parts of each channel are combined into one column.

Chan1 Real	Chan1 Imaginary	Chan2 Real	Chan2 Imaginary
-0.595855712890625	0	-0.600830078125000	0
0.0948089658499117	0.238192645720516	0.0703144902072289	0.239271367778862
-0.299856648710897	-0.796386275247523	-0.361158228931889	-0.651044291178772
-1.40238594968114	1.26725875656358	-1.10979114951594	1.20562663533569
0.275174437514480	-0.396458021702227	0.200145980616593	-0.424580003400416
-0.158337381097453	6.53691854622354	-0.434330681932207	6.40061288154617
4.64887776559003	0.255743021582746	5.00233408531421	1.02505846704051
2.11451511908438	-5.34299945781243	3.24262997628328	-6.30844389330469
1.45703142592327	-1.96183359316595	0.984374608467294	-3.16061821271093
1.04831110780329	2.95539947727775	-0.446541673373439	3.28720684289584
3.85428639257386	-3.59777930502586	3.31190026766722	-3.61395019507544
-0.844760993777662	-9.69478196083844	-1.26217437407680	-9.16363222052710

Figure 26 - DADiSP FFT Output

The DADiSP output above is formatted for easy numeric comparison; however the MATLAB output below is a bit more difficult because both the real and imaginary parts are embedded in the same field of each array.

Channel 1	Channel 2
-0.595855712890626 + 0.00000000000000i	-0.600830078125001 + 0.00000000000000i
0.0948089658488165 + 0.238192645720860i	0.0703144902060141 + 0.239271367780099i
-0.299856648710276 - 0.796386275245513i	-0.361158228931180 - 0.651044291177912i
-1.40238594967842 + 1.26725875656107i	-1.10979114951389 + 1.20562663533453i
0.275174437513418 - 0.396458021697599i	0.200145980615913 - 0.424580003396650i
-0.158337381094350 + 6.53691854620924i	-0.434330681928218 + 6.40061288153345i
4.64887776558165 + 0.255743021587134i	5.00233408530548 + 1.02505846704159i
2.11451511908627 - 5.34299945780428i	3.24262997628307 - 6.30844389329344i
1.45703142592239 - 1.96183359316615i	0.984374608467614 - 3.16061821271108i
1.04831110780474 + 2.95539947726722i	-0.446541673370004 + 3.28720684288473i
3.85428639256594 - 3.59777930502025i	3.31190026765838 - 3.61395019507012i
-0.844760993775308 - 9.69478196082761i	-1.26217437407406 - 9.16363222051583i

Figure 27 - MATLAB FFT Output

In order to calculate any differences between the two outputs the formatting has to be the same. One simple way to do this is to bring the data into Microsoft Excel 2007 and utilize the text import wizard to split the data into multiple cells.

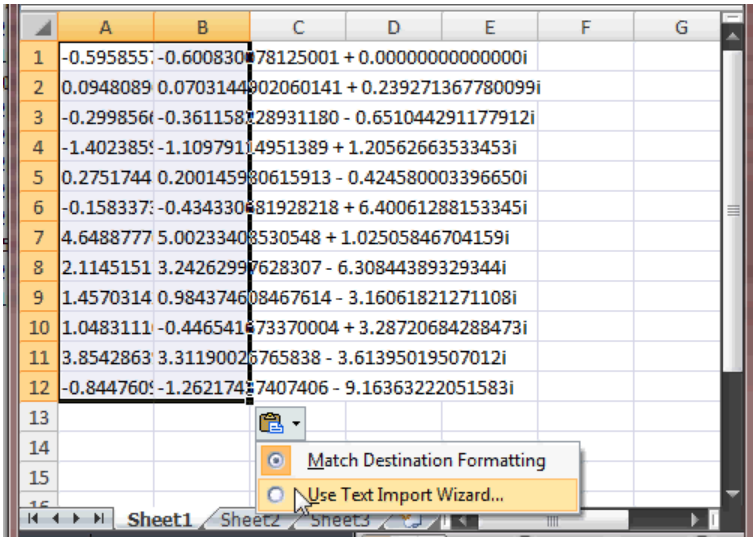


Figure 28 - Text Import Wizard

Using the default parameters of the text import wizard splits each cell into 3. So columns B and C, shown in Figure 29 can be combined to ensure that the imaginary parts have the appropriate sign and remove the "i" from each cell.

	A	B	C	D	E	F
1	-0.595855713	+	0.000000000000000i			
2	0.094808966	+	0.238192645720860i			
3	-0.299856649	-	0.796386275245513i			
4	-1.40238595	+	1.26725875656107i			
5	0.275174438	-	0.396458021697599i			
6	-0.158337381	+	6.53691854620924i			
7	4.648877766	+	0.255743021587134i			
8	2.114515119	-	5.34299945780428i			
9	1.457031426	-	1.96183359316615i			
10	1.048311108	+	2.95539947726722i			
11	3.854286393	-	3.59777930502025i			
12	-0.844760994	-	9.69478196082761i			
13						
14						
15						

Sheet1 Sheet2 Sheet3

Figure 29 - After Text Import Wizard

Figure 30 shows the MatLab Data once the formatting is complete for both channels and looks very similar to the output from DADiSP. Once that is completed for each channel, each column can be compared to its analogous array from the DADiSP output. Then the average difference, standard deviation and relative standard deviation between the signals can be calculated.

Chan1 Real	Chan1 Imaginary	Chan2 Real	Chan2 Imaginary
-0.595855712890626	0.000000000000000	-0.600830078125001	0.000000000000000
0.094808965848817	0.238192645720860	0.070314490206014	0.239271367780099
-0.299856648710276	-0.796386275245513	-0.361158228931180	-0.651044291177912
-1.402385949678420	1.267258756561070	-1.109791149513890	1.205626635334530
0.275174437513418	-0.396458021697599	0.200145980615913	-0.424580003396650
-0.158337381094350	6.536918546209240	-0.434330681928218	6.400612881533450
4.648877765581650	0.255743021587134	5.002334085305480	1.025058467041590
2.114515119086270	-5.342999457804280	3.242629976283070	-6.308443893293440
1.457031425922390	-1.961833593166150	0.984374608467614	-3.160618212711080
1.048311107804740	2.955399477267220	-0.446541673370004	3.287206842884730
3.854286392565940	-3.597779305020250	3.311900267658380	-3.613950195070120
-0.844760993775308	-9.694781960827610	-1.262174374074060	-9.163632220515830

Figure 30 - MatLab Data Formatted to Match DADiSP Data

Repeating the above steps for 9 other sample wav files showed similar success across multiple input files. Now that the outputs from MatLab and DADiSP are verified to be consistant, the SIMULINK model can be packaged for deployment and sharing.

3.4 Creating Startup Scripts

In order to make sure that the DADiSP model within simulink will start up properly and set all the necessary variables and registry settings every time it's started, I created a set of scripts that take care of everything necessary to execute the model and allow the user to simple specify settings in the model and then go. These scripts are also intelligent enough so that if the user is a new user to the computer it is being executed on, or an existing user, everything will be set appropriately without any additional steps. These are almost the most important pieces used when building an MSI later.

To start building these scripts we take all of the files we've used and compiled thus far and put them into a centralized Directory. For the purposes of this example, I've used "C:\Program Files (x86)\WPI\DADiSP_SimuLink" because I have been using a 64-bit version of Windows and DADiSP is a 32-bit application. On a 32-bit operating system, we would just use "C:\Program

Files\WPI\DADiSP_SimuLink". The files in this directory are "dadisp.mdl", "Plot_and_filter.m", "sample.wav" and "readme.txt". "dadisp.mdl" is the Simulink Model File. "Plot_and_filter.m" is the m-file referenced within the model. "sample.wav" is the wav file used within the model. "readme.txt" is a small readme file telling users not to modify the contents of this directory manually.

I've added four more files to this directory which will serve the purpose of the startup scripts. "copy_user_files.bat" will copy the necessary model files from the current directory into the users' directory under %USERPROFILE%\WPI\DADiSP_Simulink\.

```
@echo off
mkdir %USERPROFILE%\WPI\DADiSP_SimuLink
copy "%PROGRAMFILES(x86)%\wpi\DADiSP_SimuLink\dadisp.mdl" %USERPROFILE%\WPI\DADiSP_SimuLink\ /y
copy "%PROGRAMFILES(x86)%\wpi\DADiSP_SimuLink\Plot_and_filter.m" %USERPROFILE%\WPI\DADiSP_SimuLink\ /y
copy "%PROGRAMFILES(x86)%\wpi\DADiSP_SimuLink\sample.wav" %USERPROFILE%\WPI\DADiSP_SimuLink\ /y
```

The "create_user_files.vbs" will check a bit in the registry to see if the user has already copied the model files to their own user directory. If they have, then it will not execute the "copy_user_files.bat" so as not to overwrite the users' files in case they've made any changes, if they have not, then it will copy them over.

```
option explicit
on error resume next
dim oWshShell, FSO
dim struserFiles
set FSO = createobject("scripting.filesystemobject")
set oWshShell = createobject("wscript.shell")
'Tell DADiSP it is off (create's registry key if necessary)
struserFiles = 0
struserFiles = owshshell.regread("HKCU\Software\WPI\DADiSP\user_Files_copied")
If err then
'install the key and copy the files
        owshshell.run copy_user_files.bat, 0, true
        oWshShell.RegWrite "HKCU\Software\WPI\DADiSP\Files_Copied", "1", "REG_SZ"
end if
set struserFiles = nothing
set fso = nothing
set oWshShell = nothing
```

The “Reset_DADiSP.vbs” file performs a couple of functions. First it checks and resets the registry value in the user hive that is checked by the SIMULINK model to see if the DADiSP ActiveX connection has already been made. It also ensures that the directory “c:\temp” exists and is readable and writeable bay users and administrators alike. To that end, the first time that these scripts are executed, they must be executed by an administrator in order to make sure that the “c:\temp” directory is created and the security permissions are set correctly.

```
option explicit
on error resume next
dim oWshShell, FSO
set FSO = createobject("scripting.filesystemobject")
set oWshShell = createobject("wscript.shell")
'Tell DADiSP it is off (create's registry key if necessary)
oWshShell.RegWrite "HKCU\Software\WPI\DADiSP\Turned_On", "0", "REG_SZ"
'make sure the temp directory necessary exists
if FSO.folderexists("c:\temp") then
else
        fso.createfolder "C:\temp"
        owshshell.run "%COMSPEC% /c Echo Y| cacls c:\temp /t /c /g Administrators:F System:F Users:F", 0, True
end if

set fso = nothing
set oWshShell = nothing
```

Finally the “Startup DADiSP Model.bat” file brings all of these files together and creates one point of entry for anyone starting the Simulink model for the first time or for the 100th time. It executes the two visual basics scripts discussed above and then it will execute Matlab using the users’ copy of the “dadisp.mdl” file. Creating a shortcut to this icon in the start menu for all users provides it as a single point of entry to the model meanwhile ensuring that all the other various registry keys and files are set appropriately.

```
Echo Starting DADiSP/Simulink Project
@echo off
wscript "C:\PROGRAM FILES (x86)\WPI\DADiSP_SimuLink\create_user_files.vbs"
wscript "C:\PROGRAM FILES (x86)\WPI\DADiSP_SimuLink\Reset_DADiSP.vbs"
matlab.exe -nosplash -minimize -r "load_simulink;open %USERPROFILE%\WPI\DADiSP_SimuLink\dadisp.mdl"
```

3.5 Creating Installable Package (MSI)

In order to create and installable package for the DADiSP model, I used an application called Wise Package Manager from Symantec Corporation. However, there are also many other free open source software or commercial software packages available that allow creation of single Microsoft Installers (MSI) or Windows installation executables. To start creating the installer, we open up Wise Package Manager, create a new project and then open up the "Windows Installer Editor" as shown in Figure 31.

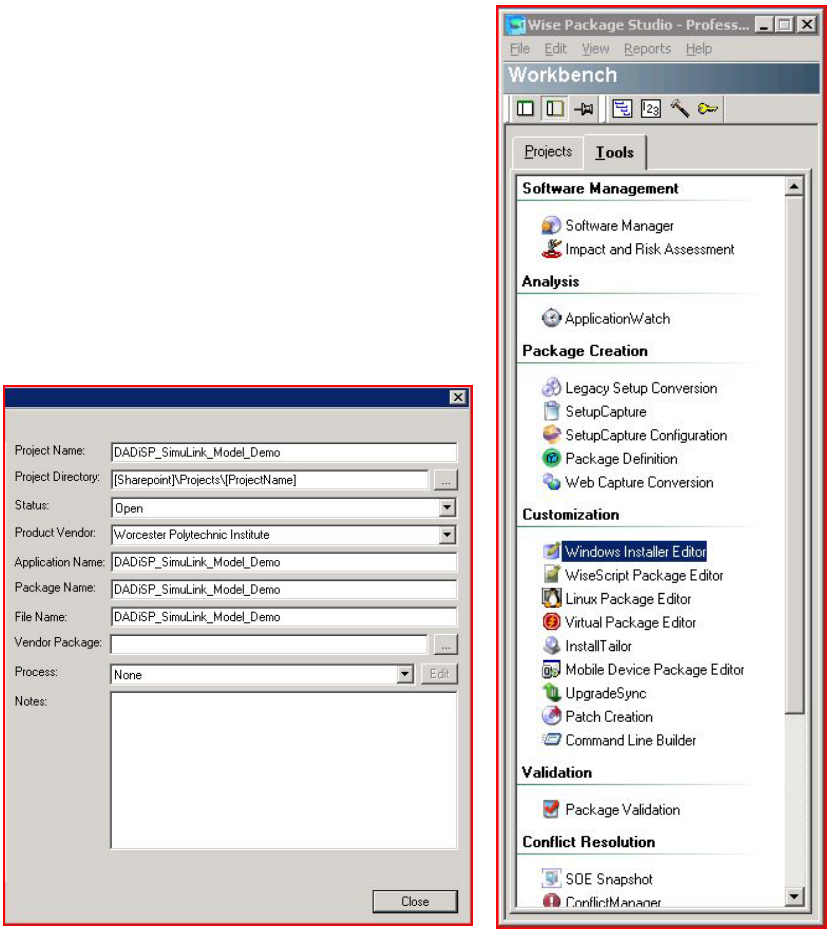


Figure 31 - New Wise Project

Next we go over to the “Files” section and we add the files that we are going to drop into the Program Files directory. Here we can also add directories as we please. This is shown in Figure 32.

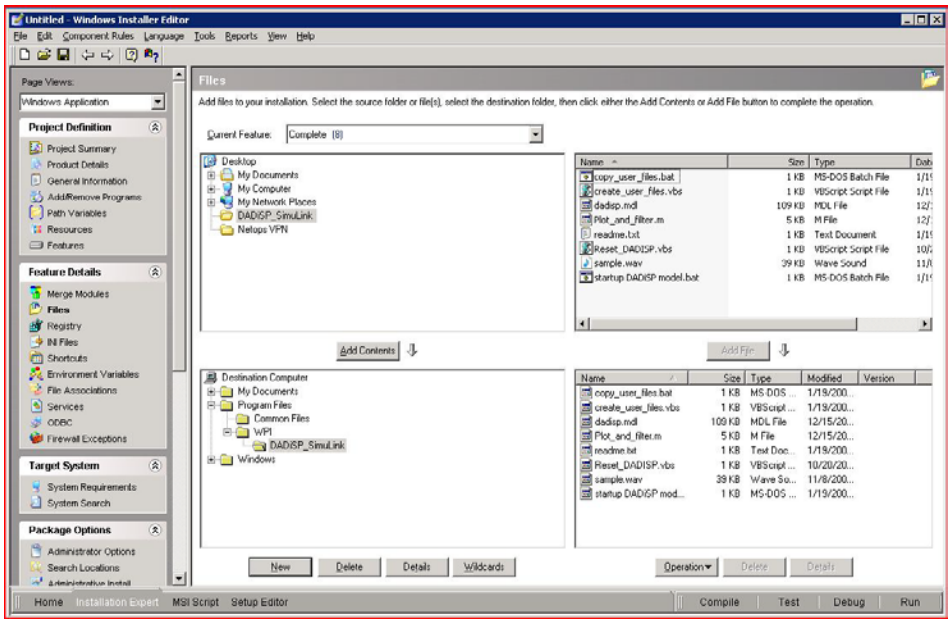


Figure 32 - Added Files

Next, set the product information and specify how we want things to look in the Add/Remove Programs menu as seen in Figure 33 and Figure 34.

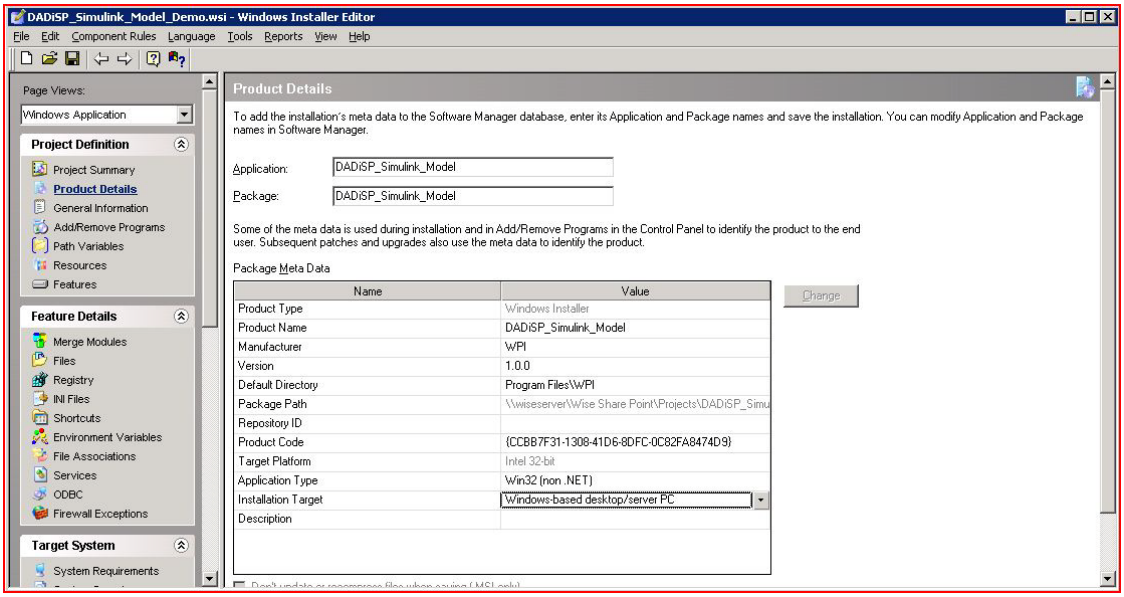


Figure 33 - Set Product Information

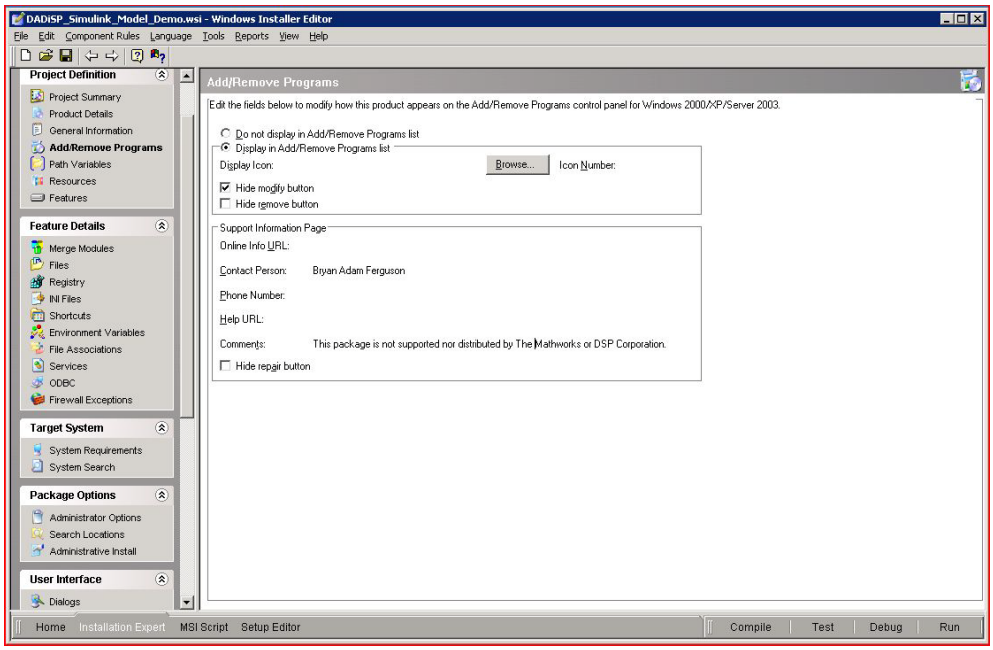


Figure 34 - Add/Remove Programs Information

Lastly, Wise package manager allows for customizable themes on the installation menus. Figure 35 shows a theme with the Worcester Polytechnic Institute seal.

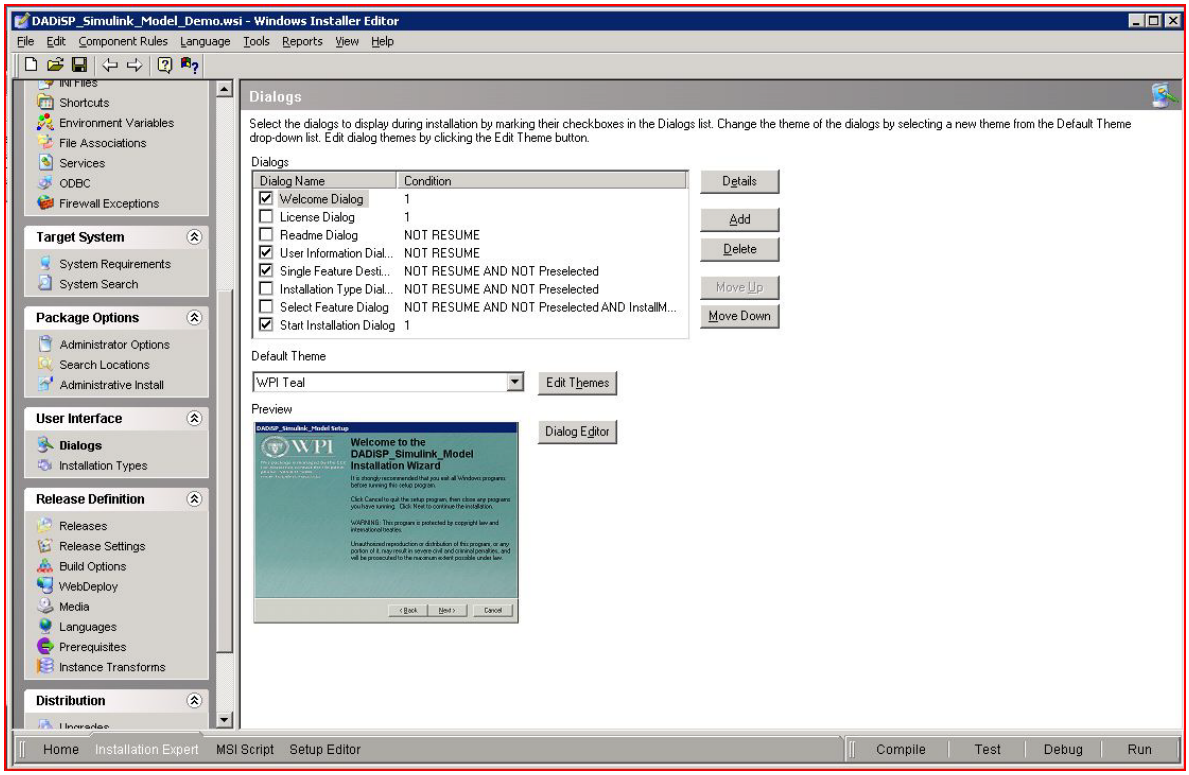


Figure 35 - Menu Themes

Finally Figure 36 shows the menu for compiling these modifications into an MSI. The MSI can be compiled on the local computer, or if using a Wise Package Server, can be compiled remotely on the server. This is useful if the server is more powerful than the computer with the Wise package Manager client.

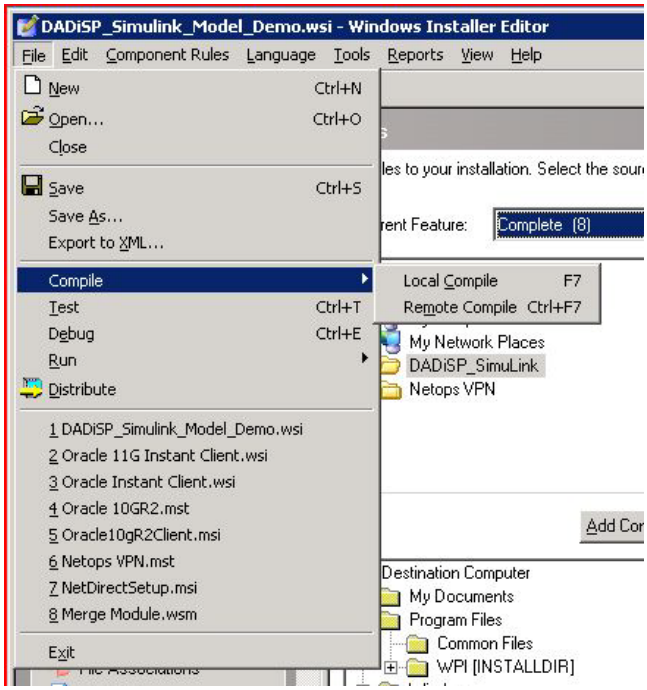


Figure 36 - Compile MSI

Once the MSI is executed, it will appear in the Add/Remove Programs Menu and any shortcuts that were specified to be installed will appear in the start menu, as shown in Figure 37 and Figure 38 respectively.

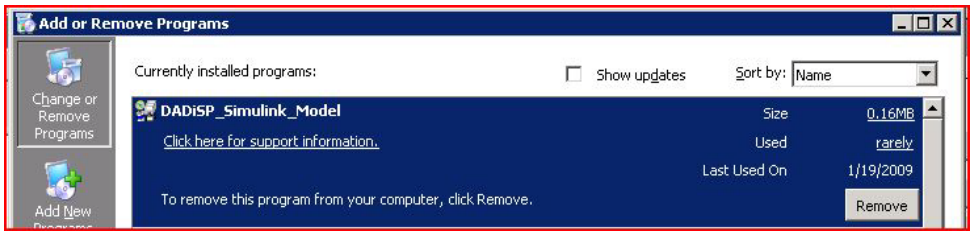


Figure 37 - Add/Remove Programs

Figure 38 shows the installed shortcut for the DADiSP SimuLink Model Demo which when selected will open up the SimuLink model that is a demo of the DADiSP functionality.

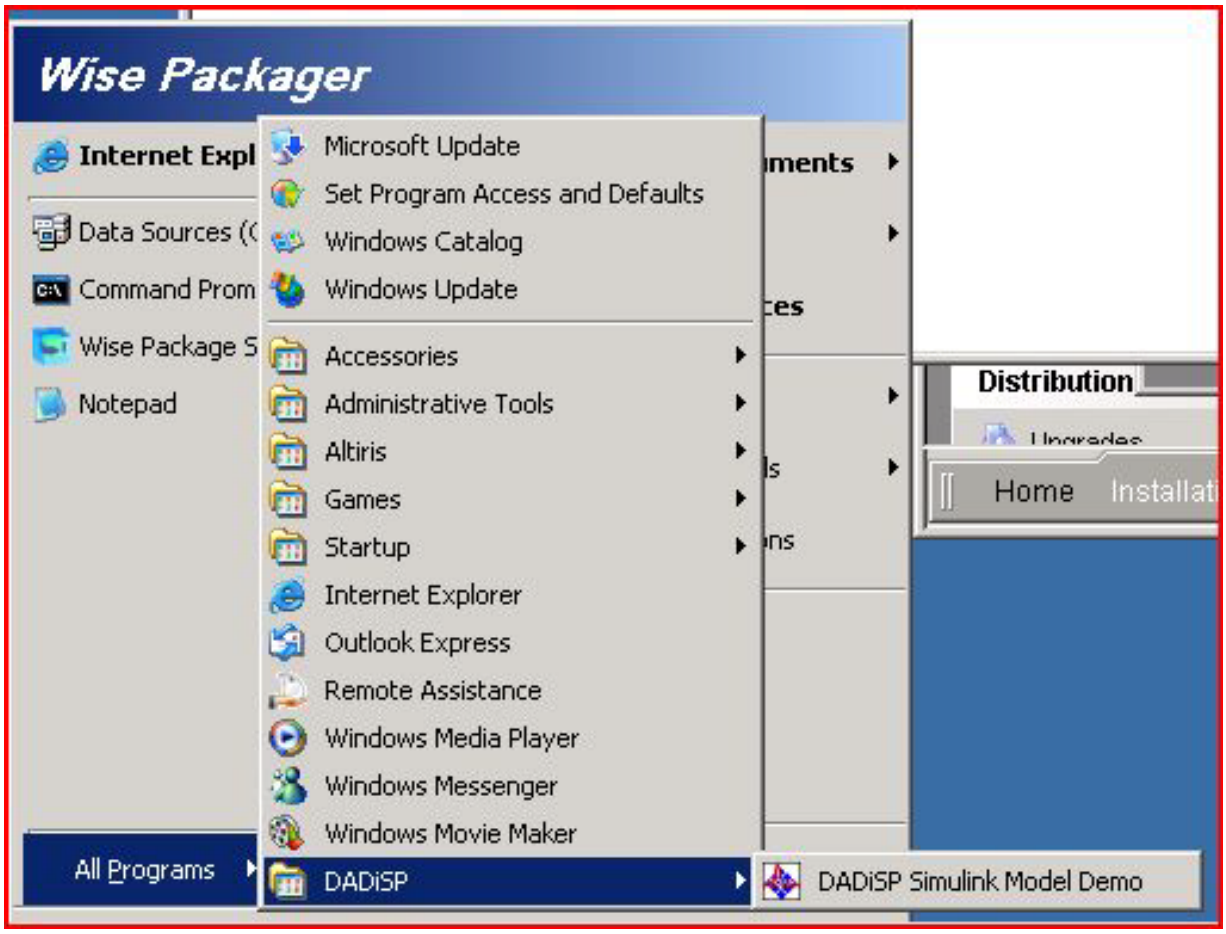


Figure 38 - Installed Shortcuts

Now that the final product is created, this single MSI can be distributed and installed on a Windows XP or Windows Vista computer. Pre-requisites, however, include MATLAB 2008a or better and DADiSP 6.0 B12 or later be installed on the computer before being able to run the DADiSP Simulink Model Demo.

4. Results & Recommendations

The sample model for DADiSP has shown a number of results some of which support that such an implementation will be useful to end users and signal processors, however other data shows that optimization and perhaps a different implementation might be necessary for everyday use. The biggest disappointment with a SIMULINK and DADiSP implementation is that speed is drastically sacrificed. However future work may be able to overcome these issues.

4.1 Speed & System Resources

The purpose of this project was to utilize DADiSP through a run-time environment, such as SIMULINK. Unfortunately, using SIMULINK to transfer data to DADiSP sample by sample results in a significant lag. This appears to be, however a result of the ActiveX connection and not a problem with DADiSP, MATLAB or Simulink.

Using SIMULINK or MATLAB to import a wav file, perform a simple filter task or calculate an FFT occurs in less than 1 second on a modern computer. Using only DADiSP to perform the exact same task also takes approximately less than 1 second on the same system. Also importing a signal into MATLAB, transferring the entire signal into DADiSP and then performing the same tasks executes in a matter of seconds. However when using SIMULINK to import a WAV file then transfer each sample one at a time to DADiSP takes multiple orders of magnitude longer, on the order of 15 minutes for fewer than 40,000 samples.

This can only be a result of overhead from ActiveX. When using the ActiveX connection only once to transfer a large block of data, the overhead is not noticed. However when executing a data transfer over ActiveX, each sample is transferred with an individual ActiveX connection as opposed to the one command, these repeated commands in multiply the overhead exponentially.

4.2 FFT Comparison

After applying the technique discussed in section 3.3 Comparing FFT Output to 10 different sample files, I was able to determine that while there is a slight difference between the computed FFT of DADiSP and MATLAB, the relative standard deviation is negligible for most applications.

Sample name	Chan1RE (%)	Chan1im (%)	Chan2RE (%)	Chan2IM (%)	AVERAGE (%)
Chime	0.00159791	0.001272093	0.002041029	1.20142E-15	0.001227758
Chord	0.009900774	0.000497626	0.004313568	0.00252236	0.004308582
Ding	0.00147216	0.00056196	0.005357983	0.001205079	0.002149295
Minimize	0.000852655	0.006122675	0.025211287	8.949E-05	0.008069027
Notify	0.01275789	0.000102854	0.009144268	0.000108322	0.005528334
Print	0.000636373	4.64845E-05	5.45696E-05	0.000450013	0.00029686
Recycle	0.09449649	0.017879374	0.137820264	0.013642558	0.065959672
Restore	0.003417942	0.007290816	0.000775511	0.011810649	0.00582373
Ringin	0.00168941	0.00035759	0.001204649	0.000704448	0.000989024
Shutdown	0.000731062	0.000629448	0.000919202	0.001189132	0.000867211

Figure 39 - Relative Standard Deviation of Difference between outputs

Figure 39 above shows the relative standard deviations in percentage of the difference between the output of the DADiSP calculated FFT and the MATLAB calculated FFT by channel as well as the average for each channel. The average relative standard deviation for each signal is less than 0.1%. The average relative standard deviation for all samples is 0.009691756%. This shows that there is an extremely close correlation between the DADiSP calculated FFT and the MATLAB calculated FFT.

4.3 Future work

The purpose of the DADiSP model is to use it as a proof of concept for additional work. While it can be used as a blueprint for to perform more advanced functions, speed issues may preclude such implementations. However, a similar methodology with another implementation may be more effective.

4.3.1 LabView

LabView might be a natural next step for an attempted implementation as LabView is a run-time environment. Additionally, LabView supports ActiveX connections and even supports using Level-1 and Level-2 S-Functions from SIMULINK, meaning that the code provided in this example may only need to be simply dropped in place of a LabView model. However because it will still utilize ActiveX from within an S-Function there may still be the same inherent speed issues as with SIMULINK.

However LabView may have an ability to interface via an ActiveX block connection that may be more effective than a custom built S-Function. Again this may result in more successful speeds, however may also result in the same speed lag as SIMULINK. A more custom solution may be more effective, however may require significant additional programming.

4.3.2 Third-Party Block Diagramming

Another option to address the issues with Speed may be to utilize a general block diagramming and run-time application. This could be either an open or closed source application as long as it supports significant programmability. However, given that the only way to interface with DADiSP from an external application is through ActiveX, the same exact lag may be encountered. ActiveX appears to be the resulting bottleneck and the limiting factor in each scenario.

4.3.3 DADiSP Application Builder

The DADiSP Application Builder (DAB), however may be a different and more viable alternative. The biggest advantage of the DAB is that it allows direct coding from any Microsoft .NET language and can be embedded like a custom control. Because it communicates directly it with .NET languages, it may overcome the speed issues that are inherint with the full DADiSP application. In addition, it also has a smaller footprint than the full DADiSP application and can be redistributed royalty free.

However the major catch with this implementation may be finding an open source run-time, block diagramming application that is built in a Microsoft .NET language. Alternatively a team could build a block diagramming run-time environment from the ground up using a .NET language and implementing the DAB natively in that application which may prove most effective and possible. However, in the end there may be one option that will ultimately be the most effective overall.

4.3.4 Direct Integration with DADiSP

What may be the ultimately the most effective option, would be to implement a block diagramming, run-time extension directly within DADiSP. Unfortunately this option is most out of the scope of an MQP because the code to DADiSP is closed source. However should DSP Development ever decide to migrate their code to an open source model, that may allow for such a possibility. However that would be a business decision of the developer. One possible model may be that the open source version is untested and released under a different name, while the commercial version of DADiSP is fully tested and verified by the vendor.

Another way that this could be made possible is if DSP Development were to decide to implement this feature via proprietary means. Again, however, this would be out of the scope of any Major qualifying project as the parts of the code would need to be available for the purposes of the project. This would be more effective than even an open source model since DSP Development developed the original application and already know how the existing code functions.

5. Conclusions

While DADiSP does not have its own real-time execution environment, it can be used as part of another environment, such as SIMULINK. This provides for block-diagram functionality and the more appealing DADiSP graphical interface and simplified tools for signal manipulation. However, doing so with current Active-X based technology comes at the cost of speed. While ActiveX has evolved over the past 15 years it is unfortunately not yet capable of handling continuous, long data sets and the future is unclear as to if it ever will.

Traditionally, the problem of slow software has been met by faster hardware, while this approach can be effective and has, until now, satisfied the prediction of Moore's Law; it isn't necessarily the correct answer to every problem. Hyper threading, multiple core processors, faster RAM, hard drives and Flash memory have all enhanced and assisted engineering problems that could in some cases be solved in software. Given the ubiquity of cloud or clustered computing, allowing an application to run across a cluster of systems may make up for the lag in transferring data between SIMULINK and DADiSP. However, again, is applying more hardware to the problem appropriate?

While this isn't to say that ActiveX may someday become more efficient to allow for rapid, continuous data point transfer, it won't be in the current and next generation of Microsoft Windows operating systems. While it may not be feasible to open the source code for DADiSP to allow a third party to develop a directly integrated block-diagramming application for it, another approach may be to develop a non-ActiveX based API. This could be based on Remote Procedure Calls (RPC) or some other, more generic protocol that perhaps might support operating systems other than Windows.

A faster or more generalized API for DADiSP may ultimately benefit the signals engineer population at large. Each generation is expected to build upon the work of the previous. As such, it is

important for the tools for the next generation to become not only more complex in functionality but simpler in execution. The purpose of this is not to allow future engineers to disregard the fundamentals of engineering, but rather to give them a boost towards further understanding. DADiSP is effective in this mission by allowing for easy visualization and manipulation of signals in ways that other software packages don't allow. By incorporating a real-time environment, a new dimension has added to enhance this experience further.

With a real-time execution of DADiSP, students can now see not only a signal as it progresses, but also how its FFT evolves as the signal is processed further, as well as how a filter may morph a signal differently as more of the signal information becomes available. This means that a real-time DADiSP is not only a tool for solving engineering problems, but also an educational tool for students. No matter who uses this enhanced addition to DADiSP be it an engineer, student or faculty, one of the most important goals can bet met; better understand. This, in the end is what allows engineers to design and innovate to the best of their ability.

References

"AMS Glossary". Glossary of Meteorology. 19 January, 2009 <http://amsglossary.allenpress.com/glossary/browse?s=d&p=28>.

"COM: Componet Object Model Technologies". Microsoft Corporation. 11, December 2008 <http://www.microsoft.com/com/default.mspx>.

"Component Object Model." Wikipedia, The Free Encyclopedia. 2 May, 2009, 17:20 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/Component_Object_Model>.

"Drag-and-drop." Wikipedia, The Free Encyclopedia. 19 March, 2009, 22:11 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/Drag-and-drop>.

"DCOM Technical Overview". Microsoft Corporation. November, 1996 <http://msdn.microsoft.com/en-us/library/ms809340.aspx>. 11 December, 2008

"DSP Development Corporation: About DADiSP". DSP Development Corporation. 4 November, 2007 <http://www.dadisp.com/aboutdad.htm>.

"DSP Development Corporation: ActiveX". DSP Development Corporation. 11/04/2007 <http://www.dadisp.com/activeX.htm>.

"DSP Development Corporation: Corporate Background". DSP Development Corporation. 4 November, 2007 <http://www.dadisp.com/corpback.htm>.

"DSP Development Corporation: DADiSP Application Builder". DSP Development Corporation. 11 November, 2008 <http://www.dadisp.com/dab.htm>.

“DSP Development Corporation: DADiSP Product Family". DSP Development Corporation. 4 November, 2007 <http://www.dadisp.com/products.htm>.

"DSP Development Corporation: Market Background". DSP Development Corporation. 4 November, 2007 <http://www.dadisp.com/markback.htm>.

"Dynamic Data Exchange." Wikipedia, The Free Encyclopedia. 5 March, 2009, 21:29 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/Dynamic_data_exchange>.

"The MathWorks - Contact Us - Worldwide Offices and Representatives". The Mathworks. 19 January, 2009 <http://www.mathworks.com/company/aboutus/contact_us/>.

"The MathWorks - Founders - Jack Little". The Mathworks. 19 January, 2009 <http://www.mathworks.com/company/aboutus/founders/jacklittle.html>.

"MATLAB - Introduction and Key Features". The MathWorks. 7 November, 2007 <http://www.mathworks.com/products/matlab/description1.html>.

"MATLAB News & Notes - Summer 1998 - MATLAB Compiler and New Support for ActiveX in MATLAB 5.2". The MathWorks. 7 November, 2007<http://www.mathworks.com/company/newsletters/news_notes/sum98/sum98activex.html>.

“Mono". Mono Project. 19 January, 2009 <http://www.mono-project.com/Main_Page>.

"Object Linking and Embedding." Wikipedia, The Free Encyclopedia. 30 April, 2009, 11:48 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/Object_Linking_and_Embedding>.

"OLE Automation." Wikipedia, The Free Encyclopedia. 28 March, 2009, 20:17 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/OLE_Automation>.

Race, Randy and Rabin Tamang. Personal INTERVIEW. September 20, 2007

"Simulink - Introduction and Key Features". The MathWorks. 7 November, 2007 <http://www.mathworks.com/products/simulink/description1.html>.

"Structured Storage." Wikipedia, The Free Encyclopedia. 7 February, 2009, 3:26 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/Structured_storage>.

"Video: The Origins of MATLAB". The Mathworks. 19 January, 2009 <http://www.mathworks.com/company/aboutus/founders/clevemoler.html>.

"Visual Basic Extension." Wikipedia, The Free Encyclopedia. 26 December, 2007, 16:20 UTC. Wikimedia Foundation, Inc. 3, May 2009. <http://en.wikipedia.org/wiki/Visual_Basic_Extension>.

"Z-transform." Wikipedia, The Free Encyclopedia. 1 May, 2009, 2:10 UTC. Wikimedia Foundation, Inc. 1, May 2009. <http://en.wikipedia.org/wiki/Z-transform>.