TechTools

DigiView Plug-in Guide

DigiView Plug-in Guide

Printed: April 2016 in Rowlett, Texas U.S.A.

Publisher

TechTools
P.O. Box 2408
Rowlett, TX 75030-2408
U.S.A.

Technical Sales

(972) 272-9392

Fax

(972) 494-5814

Email

Sales@tech-tools.com

Support@tech-tools.com

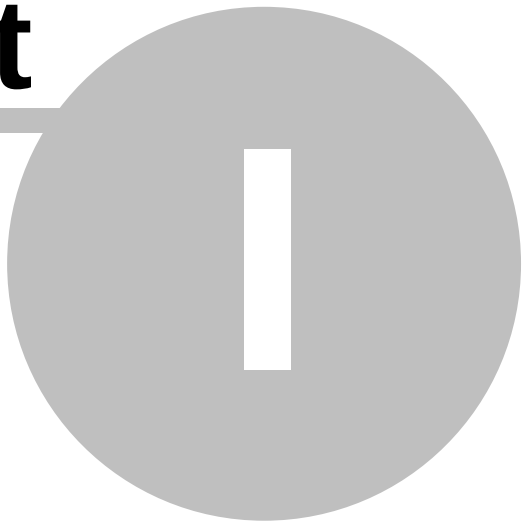
On The Web

www.tech-tools.com

Table of Contents

Plug-in Overview

Part



1 Plug-in Overview

Plug-in Developer's Guide

(PDK version 1.2.1)

Plug-ins are user created extensions to the DigiView application. They allow the user to modify the formatting of DigiView's built in interpreters, implement entirely new custom protocols and/or control the run-time behavior of the application.

Plug-ins are fully integrated into the DigiView applications. Signals based on plug-ins can be searched,exported, and printed in all the same manners as built-in types. All snaps, scrolls, lists, waveform views, searches, auto-searches, etc work in exactly the same way as built-ins. In fact, the internal protocol interpreters use the same framework as the plug-ins, ensuring equal functionality.

Plug-ins can be written in any language and do not require any special Windows programming knowledge. They are written as simple console-mode executables or scripts, using standard console read and write calls to interface with the application. We provide wrapper routines so that you do not have to deal with the read/writes at all. You simply respond to calls to 'parse()' to accept data and use calls like 'startfield()' and 'stopfield()' to return your data and formatting instructions. No knowledge of DLLs, sockets, COM, OLE, etc. is required. In fact, you don't need to know anything at all about Windows specific programming.

The 'Getting Started' tutorial demonstrates that you really can create your first functional plug-in in less than 20 minutes!

1.1 Terminology

Plug-ins have many uses ranging from serial protocol analyzers to soft triggers. Each application might have different terms for the data generated. We will use the following terms throughout this discussion.

Samples:

The raw data gathered from the hardware at its sample rate.

Channels:

These are the physical connections to the target. Our Logic Analyzers have 9,18 or 36 channels.

Active Channels:

The physical channels that are assigned to active signals. These are the channels the hardware is monitoring.

Signal

A higher level abstraction. It maps physical channels to specific purposes in the signal. All displays, searches, triggers, etc are defined in terms of Signals; not channels. You can reassign a signal to a different channel without changing anything else. Multiple signals can use the same channel where appropriate (e.g.: several SYNC signals could use the same channel for their CLOCK function.)

Signal Parser

This refers to the routines used to translate the raw captured data into the representation in the waveforms and list views. The signal parser uses the channel mapping and signal configuration options to extract data from the raw capture data, interpret it and format it for display.

Pre-processor and Post-processor

All signal parsers consist of 2 parts; a pre-processor and a post-processor. The pre-processor interprets the raw capture data and sends this information to the post-processor. The post-processor analyzes this data to generate the display formatting, colors and framing.

Event:

The output from the pre-processor (input to the post-processor) is called an EVENT. Events consist of a time-stamp, some data and possibly some flags. These represent higher level activities than raw signal transitions. Typical events will indicate errors in the protocol, start and stop framing (if part of the protocol), a completed field of data or perhaps a single bit of data. The exact contents of an event vary with each pre-processor.

RawData Events:

Similar to events, except the data portion of the event contains the raw channel levels at this timestamp, rather than processed data from a pre-parser.

Field:

The final post-processor outputs a series of field definitions. Fields are stored in the signal's internal state table. A field definition represents a single cell of data. It is displayed as a rectangle with its value printed inside. In some serial protocols, the field widths could vary. In

others, they are consistent. In the basic ASYNC interpreter, each character is a field. In the STATE interpreter, each STATE is a field. In I2C, there are a number of predefined fields of varying length.

Frame:

Some protocols group fields into Frames (sometimes called packets). A frame might represent a complete command or transaction. In other cases, the data might be arbitrarily grouped into fixed length pieces for easier viewing. We display a FRAME as a series of connected fields with the first field starting with '<' and the final field ending with a '>'. In I2C, the frame is delimited by specific start/stop conditions on the physical lines. Other systems might use sync signals, field counts, timeouts, or specific characters to mark frame boundaries. Frames' start/end conditions are specially tagged/formatted fields.

1.2 Types of Plug-ins

DigiView supports 3 types of plug-ins; mini, full and hybrid.

Mini Plug-ins

Mini Plug-ins use one of the built-in parsers as a pre-parser, simplifying your work. The pre-parser handles the low level details of extracting the link level information. Your plug-in can concentrate on higher level issues like formatting, adding another level of protocol, soft triggering or filtering. The plug-in depends on the pre-parser supplied user options for the basic protocol configuration. The mini plug-in can add additional options if needed (see the I2C plug-in example) but can not add new channelselect options. For example, if you have a custom protocol implemented over an ASYNC link, you could write a mini-plug-in based on the internal ASYNC pre-parser. The pre-parser will extract the ASYNC characters for you (like a UART would). Your plug-in would inspect the characters and look for your protocol's commands, parameters and any framing indications. Your plug-in would then display the protocol as you see fit.

Full Plug-ins

A full plug-in is based on the RAW data pre-processor. The RAW pre-processor simple filters out all data samples that do not involve a transition on one of the channels your plug-in is monitoring. It does not provide any user configurable options. All user options for the protocol (including channel-selects) are specified by the plug-in. The plug-in is responsible for all low level interpretations of the signal changes. It looks for bit timing, enable levels, clock edges, etc. and determines what they mean.

Hybrid Plug-ins

A hybrid plug-in is based on an internal pre-parser like the mini-plug-in. However, it also specifies additional channels to watch and assumes all responsibility for them. In this configuration, DigiView sends the plug-in all of the events generated by the pre-parser as well as raw data events whenever one of the additional monitored channels transition. The pre-parser events and raw data events are properly time sequenced. A possible use for a hybrid plug-in might be to add a unique framing signal or additional control signals to an existing built-in protocol. For example, you might be sending ASYNC characters across a half-duplex bus. Your plug-in could monitor the DIRECTION control line and adjust the display formatting to differentiate which end of the link sent the message. The 'HalfDuplex' example plug-in demonstrates this.

1.3 Plug-in Capabilities

Plug-ins can extend the DigiView application in a number of ways:

Modify formatting

The 'echostate' example demonstrates a functional Plug-in in 24 lines of code. It simply displays state fields in a different color. This is the most basic operation a Plug-in could do; change the way the data looks. A Plug-in could also change what is printed in the field as easily. For example, it could easily substitute the text 'A/D' every time it sees the value '0x10' in a particular field.

Add parameters or control signals to an existing protocol

Plug-ins can extend an existing protocol by adding extra control signals or parameters. The 'HalfDuplex' example demonstrates adding a direction line to the ASYNC parser. This would extend the ASYNC parser to support a half-duplex bus (where a control signal switches the bus alternately between IN and OUT directions.) Several of the Plug-in examples add a 'SHOW FIELD IDLE' parameter, controlling whether idle periods should be shown between fields.

Add Protocol Layers to existing parsers

Protocol layers can be very simple or complex. A simple protocol layer might involve just adding framing. Look at the 'FrameChar' for an example. It adds a framing level to the basic built-in ASYNC parser. Whenever it sees a specific character, it starts a new frame. It also watches for an escape character to allow the start-of-frame character to occur in the data payload. A more complex protocol layer might include interpreting the first field of the frame as a command and the balance as command-specific parameters.

Add entirely new protocols

Using a full Plug-in, you could implement new protocols from the link level up. You have full access to everything captured (related to your Plug-in). You can watch as many channels as you want and interpret them in any way you want. For example, we currently do not have built-in CANbus interpreter, but it could be implemented as a Plug-in. CANbus is different enough from the built-in parsers that it would have to be built as a full Plug-in. The Plug-in would need to do the async bit timing to extract the link level bits and then combine the bits into CANbus specific fields.

The track2full and full-DAC8045 examples demonstrate simple protocols developed with full parsers. These particular ones could have been based on built-in preparsers but we chose to implement them as full, raw parsers.

Analyze the data contents and/or timing

Plug-ins can evaluate the field values while it is generating field information. It can generate text ('PARITY ERROR') in place of field values. It can check protocol specific sequences and print errors for field values ('ILLEGAL NAK'). Plug-ins can verify timing (down to the logic analyzer's sample rate) and report the results as field values.

The ASYNCWD example demonstrates adding 'TIMEOUT' fields to the data stream when it detects too long of an idle between characters.

Control DigiView's run-time behavior

In addition to or instead of printing errors or timing information as field values, the Plug-in can send control fields to the DigiView application to force a save of this capture to disk, veto any default save, and/or halt an auto-run sequence. This allows the plug-in to operate as a soft-trigger, operating at the protocol level and/or as a filter to automatically sort through a sequence of captures.

The ASYNCWD example demonstrates generating HALTs, FORCED-SAVES or VETO-SAVES when it detects too long of an idle between characters.

1.4 Implementation

Whenever DigiView completes a new capture, the data is transferred to a buffer on the PC. Each signal (native or plug-in) then parses the raw data into its own state table for use by all search, snap, print, export and display routines,. The state table includes timestamp, data, and formatting instructions.

The built-in signal parsers are partitioned into two parts: the pre-processor and the post-processor. The pre-processor looks at the raw data and extracts the link-level protocol. It sends protocol-specific 'events' to the post-processor. The post-processor interprets the 'events' and generates the timestamp, data, and formatting information stored in the state table.

Conceptually, plug-ins replace the post-processor. They define which pre-processor to use and then assume responsibility for the post-processing.

For example, the pre-processor for an ASYNC signal, does bit timing analysis to find the character start, data, parity, and stop bits. It also detects parity errors, framing errors, and break conditions. It then forwards DATA, 9BITADDR, END, BREAK, PARITY ERROR and FRAME ERROR events to the post-processor. The post-processor adds display formatting to this information and stores it in the state table.

A mini-plug-in based on the ASYNC pre-processor would receive the same events and send back its own data and formatting information. It could simply change the way the data is displayed. It can also implement higher levels of protocol. For example, the plug-in could look for start-of-frame characters to indicate a new packet, translate specific field values to text (0x55 = STOP) or implement in-band escaping.

Full plug-ins operate the same way except they specify 'RAW' for the pre-processor. This is a special processor that does not extract any link level information for the plug-in. It simply filters out unrelated samples from the raw data. It forwards only data samples in which one or more of the channels used by the plug-in transition. The only event forwarded by the RAW pre-processor is 'DATA'.

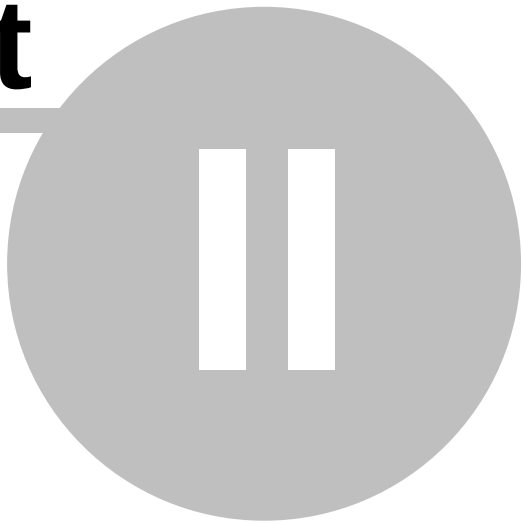
References:

Available Pre-Processors and the events they generate

Detailed Framework Documentation

Getting Started

Part



2 Getting Started

This guide is designed to help DigiView users get started developing their own custom plug-ins. DigiView users range from pure hardware types to pure software types and every mix in between. Many users have no experience compiling code on the PC while others do nothing else. This guide attempts to minimize the learning curve for all users, regardless of experience.

This section provides a basic tutorial demonstrating building, installing and verifying plug-ins. It starts by building one of the example plug-ins and installing it in DigiView. Next, it walks you through creating a new plug-in based on an existing example. Finally it walks you through customizing that plug-in with new framing and formatting configuration options.

You should have your first basic custom plug-in operational in less than 20 minutes (after installing the tool chain if needed.)

The tutorial seems more involved than it is because we go into extreme detail to assist the absolute beginner. For example, we list 5 steps to copy one file to another. We tell then HOW to copy, rather than just telling you to do it. Experienced programmers would probably just read the section and then do it their way.

There are, of course, several ways to do most things in Windows. Keeping the beginner in mind, we chose to use menu operations rather than the command line or short-cut keys. We also decided to perform all operations within the Visual Studio application rather than File Explorer.

Through-out this Tutorial we will refer to navigating to the <PDK> directory, and 'Launching' the **tutorial.dvdat** file or the Visual Studio Express **Solution** file. There are several ways to do each. To avoid repeating ourselves, we will just refer you back to this page and let you decide which method you prefer.

Navigate to the <PDK> directory:

If you accepted the default path during installation, the PDK directory will be located at:

<documents>\TechTools\DigiView\PDK-1-2

Note: Each PDK release is stored in its own directory, based on the version number. Version 1.2 is stored at PDK-1-2. Likewise, version 3.0 would be at PDK-3-0. You can go directly to the PDK directory by:

- selecting 'Start->TechTools->DigiView PDK->Browse PDK Files'
- or by double-clicking on the 'Browse PDK Files' shortcut on your desktop.

DigiView Plugin Directory:

DigiView looks for plug-ins in:

<documents>\TechTools\DigiView\plugins

Launch Solution file (CPPExample.sln):

We included a 'Solution' file for Visual Studio express. This contains predefined projects for all of the plug-in examples. We also extend it in the tutorials to include the tutorial project. There are several ways to launch this file:

- Click Start->TechTools->DigiView PDK->CPPExamples.sln
- or Click on the CPPExamples shortcut on your desktop
- or Open Visual Studio Express and select Open->solution. Then navigate to the <PDK Directory> directory and select CPPExamples.sln.
- or Navigate to <PDK Directory> and double-click on CPPExamples.sln

Launch Capture fIle (tutorial.dvdat):

We include a DigiVIew capture file containing a simple state capture. We use it in the tutorials to demonstrate plug-in functionality. There are several ways to launch this file:

- Click Start->TechTools->DigiView PDK->tutorial.dvdat
- or Click on the tutorial.dvdat shortcut on your desktop
- or Open DigiView and select 'Select File to Open'. Then navigate to the <PDK Directory> and Double-click on 'tutorial.dvdat'.
- or Navigate to <PDK Directory> and double-click on tutorial.dvdat

2.1 Install Tool Chain

You can develop plug-ins with the language and tool-chain of your choice, but all of the examples and this tutorial were developed in C++ using with the freely available Microsoft Visual Studio Express 2010 tools. We chose these tools for the examples because they are free and functional, ensuring you COULD develop plug-ins without additional expense. It also ensures that you start with known, functional examples. All documentation assumes use of these tools.

You can download the Visual C++ Express or Visual Studio Express 2010 from:

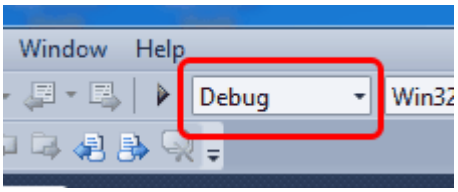
http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express

Of course, if you are an experienced Windows programmer and you already have your own preferred tool chain, you are free to work with it instead. However, you might want to use the Visual Studio Express tools to walk through the tutorial before starting your own development.

2.2 Build an Example Plug-in

To establish a working baseline, rebuild one of the provided examples as follows:

1. Launch CPPExamples.sln in Visual Studio Express.
2. Set the Solution Configuration to 'debug'



3. Select '**View->Solutions Explorer**' if not already visible.
4. Right-click on the echotest project and select **'Rebuild'**.
5. Check the 'Output' window and verify it says '0 failed'.

2.3 Install and Verify a Plug-in

Install a Plug-in

There is no formal installation procedure for DigiView Plug-ins. Simply copying a plug-in into the <DigiView Plugin Directory> will make it available to DigiView.

The echostate.exe plug-in should already be installed because the example projects were configured to auto-install the plug-ins after each successful build.

Verify the Plug-in

In order to test the plug-in, we need some capture data for it to decode. We placed an example capture file in the <PDK Directory>directory for this tutorial. To verify the plug-in works, follow these steps:

1. We do not need hardware for this tutorial so unplug your DigiView if present. This simplifies situations where your DigiView model has fewer channels than the one we used to capture this data.
2. Launch tutorial.dvdat in DigiView. Notice that we have already defined a signal using the built-in STATE parser/signal type.

3. Create a new signal based on the echostate.exe plug-in.
    - click 'Config -> Signals'
    - Click on 'echostate.exe' in the signal type list
    - Click 'Add'
4. The signal's editor should open. Set configuration options to match the STATE signal's settings. In particular:
    - Select 'rising edge' for the Clock On
    - Select Channel 4 for Clock Channel
    - Select Channels 3-0 and UN-Check 'INV' for Data Channels
    - Check 'DIS'able boxes for 'Enable Channel' and 'Frame Sync Channel'
    - Click on OK. They should now decode the same. Click 'OK' to close the Signal Editor.
5. Click OK to close the Project Settings.
6. The echostate.exe signal and the original STATE signal should look similar. They should decode the same values, but they are displayed in different colors.
7. If they don't match, then STATE signal options have changed. Double-click on the STATE signal name to open its editor and verify the following settings:
    - All of the settings described in step 5 match
    - States per Frame set to 0
    - Show Field Idles is UNCHECKED

Notice that the built-in signal (STATE) displays more configuration items than the plug-in signal (echostate). The common configuration items are the ones provided by the STATE pre-processor. The extra items seen in the STATE signal are the items provided by the built-in post-processor. Since your plug-in replaced the post-processor, these options are no longer present. Your plug-in is taking responsibility for these functions.

We will expand on this plug-in and add some custom configuration options in the next section: Create a New Plug-in.

2.4 Create a New Plug-in

In this portion of the tutorial, we will create a new plug-in, based on an existing example. In the next section we will extend it with a couple of custom configuration options.

Create a New Project:

1. Launch CPPExamples.sln in Visual Studio Express.
2. Select **View->Solution Explorer** (if it is not visible)
3. Right-click on the first line (Solution 'CPPExamples....)
4. Select **Add -> New Project**
5. Select **Win32 Console Application**, enter 'tutorial' for the **project name** and press **OK**
6. Select **NEXT** and select Application Type: **Console application**
7. Select **Empty Project** then **finish**
8. Right-click on 'tutorial' project and select **'Add -> New Item'**
9. Select **'C++ File(.cpp)**' and change the name to 'tutorial'. Click **'Add'**

Copy Existing Example into New Project

1. Expand 'echostate' and then 'Source files' in the Solution Explorer
2. Double-click on echostate.cpp to open it in an editor
3. Select **'Edit->Select All'** then **'Edit->Copy'** to copy entire file.
4. Click on the tutorial.cpp tab to activate it.
5. Select **'Edit->Paste'** then **'File->Save tutorial.cpp'.**

Change Project Settings to Auto-Install Plug-ins after each successful build:

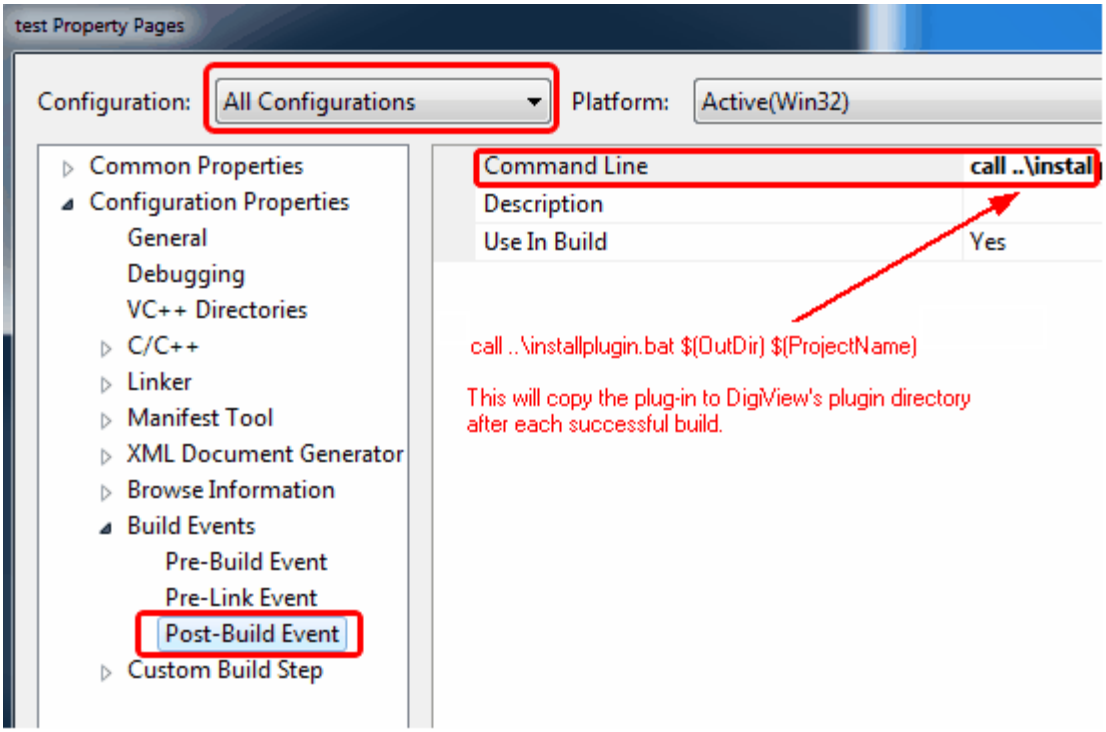
This is optional but convenient. Otherwise you will need to manually copy the plug-in from either <PDK Directory>\debug or <PDK Directory>\release to DigiView's plugin directory each time you rebuild.

1. Right-click on the project 'tutorial' and select 'Properties'

2. Change **'Configuration'** to **'All Configurations'**

3. Paste the following line into the '**Post-Build Event command line**' field and press '**OK**':

   call ..\installplugin.bat $(OutDir) $(ProjectName)



Build, Install and Verify

1. Right-click on project 'tutorial' and select 'Rebuild'. You should have a functional plug-in.

2. If you receive an error like: ' LINK : fatal error LNK1123: failure during conversion to COFF: file invalid or corrupt', you have a known issue with .Net 4.5. One work-around is to right-click on 'tutorial' and select properties. Then select 'Linker->general->Enable Incremental Linking' and set it to NO. Other work-arounds can be found with a web search.

3. Install and verify the new plug-in as in Install and Verify a Plug-in. Of course you will substitute 'tutorial.exe' for 'echostate.exe' in the instructions.

4. The tutorial.exe plug-in signal should look identical to the echostate.exe plug-in signal.

2.5 Customize the Plug-in

In this portion of the tutorial, we will extend the plug-in created in the last section. We will add a couple of configuration items and then modify the parse() routine to honor them.

Modify the plug-in

1. Disable the tutorial.exe signal in DigiView and **close the signal editor**.
2. Launch CPPExamples.sln in Visual Studio Express.
3. Double-click on 'tutorial -> Source Files -> tutorial.cpp' in the Solution Explorer.
4. Edit the body of the source code to match the code below. The changed lines are shown in RED.
5. Right-click on 'tutorial' and select 'Rebuild'. You should have a functional plug-in.
6. If you receive an error about file copy failing or access denied, the old plug-in version is probably locked. Review Disable and Task Manager. Once the old plug-in is released, rebuild the new version to invoke the copy operation again or manually copy it to the DigiView plugin directory.
7. The newly compiled plug-in should have replaced the older version and is ready to use.
8. Click on the signal name 'tutorial.exe' in DigiView's waveform display to open the signal's editor.
9. The signal editor will appear and the signal will be auto-enabled. Notice that there are 2 new configuration options. Play with them and notice the effects are immediately displayed in the waveform preview.
10. Click 'OK' on the signal editor and notice the changes to the 'tutorial.exe' waveform.

Changes to tutorial.cpp. (new/modified lines are in RED)

```
#include "../Cmdparser.cpp"

// our configuration items
static bool doframe;
static bool showblue;

void OnLoad() {}       // no one-time initialization needed
void SetInitItem(unsigned char ID, unsigned char subID, int value) {}

void SetCfgItem(unsigned char ID, unsigned char subID, int value) {
  if (ID == 0) doframe  = (value == 1);
  if (ID == 1) showblue = (value == 1);
}
void StartOfData() {}  // no start-of-parsing initialization needed
void EndOfData() {}    // no end-of-parsing finalization needed
void OnUnLoad() {}     // no final cleanup needed

void GetStrList(int ID, vector<string> &strl)
{      switch (ID)
       {
       case 0: strl.push_back("My first Custom Plug-in V.0001"); break;
       case 1:
              strl.push_back("Frame on 0 or 8,checkbox,0");
              strl.push_back("Field  Color:,radio,0,YELLOW,BLUE");
              break;
       case 2:
              strl.push_back("State,YELLOW,BLACK,{}");
              strl.push_back("State2,BLUE,WHITE,{}");
              strl.push_back("SOF,GREEN,WHITE,SOF");
              break;
       case 3:  strl.push_back("STATE"); break; // use STATE pre-parser
       case 4:  strl.push_back("1");    break; // require framework V 1
       }
}

void Parse(int64 timestamp, Data64 data)
{  // echo all data events as individual formatted fields
  if ((data.bytes[6] & 0x80) == 0x80)  //is a data event
  { if (doframe & ((data.lowint == 0)|(data.lowint == 8)))
       StartFrame(timestamp,data,2);
    else if (showblue) StartField(timestamp,data,1);
    else StartField(timestamp,data,0);
  }
}
```

Explanation

We modified the GetStrList() case 1 so that we could tell the DigiView application about the 2 new options we wanted presented to the user. See Configuration Editors for complete documentation on the available editors, their syntax and return values.

Next, we modified the SetCFGItem() call so the DigiView application could tell us what options the user chose. Notice that the SetCFGItem ID parameter tells us which configuration item we are receiving. These correspond to the order we specified the configuration items in the GetSTRList() case 1 above.

We also modified the GetStrList() case 2 to define 2 additional format strings. Now we have 3 different formats we can use. See Field Formats for complete documentation of field format strings.

Finally, we modified Parse() to use the new configuration information to format the display.

Conclusion

We extended the plug-in to allow the user to change the field color and to choose whether or not to frame when it sees a 0 or 8. We also substituted 'SOF' for the actual data in the start of frame field. This added only about 12 lines of code and the source to the entire plug-in still fits on one page.

2.6 Next Steps

You can create a lot of useful plug-ins by extending the existing examples. The configuration sections can generally be debugged without the use a debugger and simple parse() routines can often be debugged through the visual feedback provided by DigiView. The plug-in can even write debug message to the waveform.

However you will need a deeper understanding of how the framework works in order to write and debug more involved plug-ins. The remainder of this guide provides that information.

Development Tips provides a lot of useful tips for generating the formatting you want and for understanding when and why the DigiView application calls the various routines in your code.

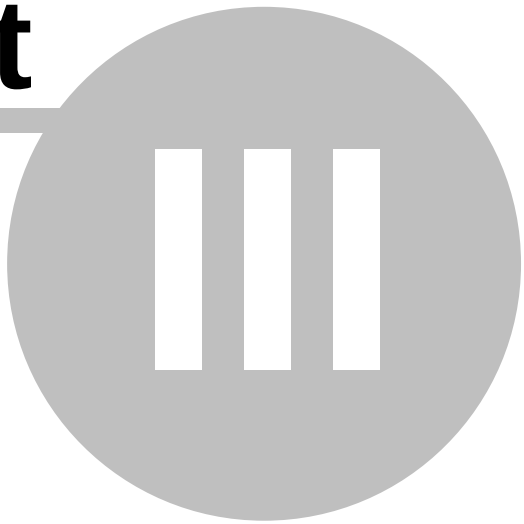
Debugging explains how to setup Visual Studio to debug the plug-in and provides some recommended work-flows to make debugging go smoothly.

Plug-in Framework explains the purpose and syntax of every file and procedure in the framework.

Plug-in Examples provides a brief description of each of the provided plug-in examples.

Development Tips

Part



3 Development Tips

This section provides a lot of useful tips for generating the formatting you want and for understanding when and why the DigiView application calls the various routines in your code.

3.1 Plug-in Dataflow

Each time the data changes (from a new capture or loading a previous capture), Each defined signal parses the data into an internal state table to be used by all search, export, print and display routines. Signals that use plug-ins, stream the data through the plug-in before storage. Multiple signals can use the same plug-in. Therefore, each signal sends its configuration to the plug-in before each parse run. The configuration items are defined by the parser and independently selected by the user for each signal.

On Signal Create

When the user defines a new signal, the first choice he makes is to select the signal type. They are presented with a list of signal types that include our native types (bus & boolean), our built-in parsers and all of the plug-ins found in our plug-in directory.

If the user selects a plug-in the following happens:

- The plug-in is loaded into memory if it is not already loaded
- The plug-in's OnLoad() routine is called if it was not already loaded
- The plug-in's GetStrList() is called once for each string list
- The user is presented with a signal editor dialog to configure the signal
- The user's selections for THIS signal are stored
- The steps in On New Data are run

On New DATA

Each time the capture data changes, the following happens:

- the plug-in's SetInitItem() is called multiple times to set some globals
- the plug-in's SetCfgItem() is called multiple times to configure the plug-in with the user's settings for THIS signal
- the plug-in's StartOfData() is called to allow final preparations

- Data events from the pre-processor are streamed to the plug-in's Parse() routine.
- Parse() uses calls to various Data Output Routines to stream back field information.
- When all of the events have been sent, the plug-in's EndOfData() is called.
- If multiple signals are using the same plug-in, the above sequence will be repeated for each signal in turn (using that signal's specific configuration and the resulting event stream)

On Signal Disable

Anytime a signal is disabled, if it is the last signal using the plug-in, the plug-in's OnUnload() is called and then it is unloaded from memory. However, its configuration is remembered.

On Signal Delete

Anytime the signal is deleted, if it is the last signal using a plug-in, the plug-in's OnUnload() is called and then the plug-in is unloaded from memory.

On Signal Enable

Anytime a signal is enabled, it is loaded into memory as described above but its editor is not invoked. The stored configuration is used. A complete parse cycle is run as described in On New Data.

On Configuration Change

Each time the user changes a configuration option (in the signal editor), a complete parse cycle is run as described in On New Data . This gives immediate feedback about the effect of their selection.

3.2 Enable/Disable

Each time you recompile your plug-in and wish to test it, you need to copy it to the plug-in directory. If DigiView is still running and your old plug-in is still in use, Windows will not let you over-write it. To release the old version, you need to do one of the following:

- shut down DigiView, copy the plug-in, restart DigiView
- delete the signal using your plug-in, copy the new plug-in, recreate the signal
- disable the signal using your plug-in, copy the new plug-in, then enable the signal

The disable/re-enable is the best. It is fast and easy and still preserves all configuration items.

You can disable/enable the signal from the Signal Definitions tab in the project settings window or from the signal's configuration editor. The signal's editor can be opened from the Signal Definitions tab or by clicking on the signal name in the waveform view.

NOTE: If you use the enable/disable checkbox from the signal editor, be aware that any changes you made to the plug-in's configuration options will NOT be reflected in the editor until it is closed and reopened. Also, anytime you make changes to any of the configuration option definitions, you should check your settings in the signal editor. If you change the option name/label, it will be treated like a new option and set to its default settings. Also, we store configuration as indexes into the options you provide so if you change some of the parameters, we could be selecting a different setting now.

3.3 Streaming and Context

It is important to realize that data is streamed to your plug-in and it is expected to stream back its data. Your plug-in will not have random access to the data or even know how much data it will receive. Since your plug-in is sharing the system with a couple dozen other plug-ins, you do not want to absorb the data into an internal array, process it and then send back your results. This is very inefficient in resource usage and is slow. The plug-in must process a single piece of data at a time and immediately return. This requires a certain mindset during development.

The plug-in must maintain its state in the protocol while parsing so that it can interpret each piece of data in context. The example 'RAWSTATE.CPP' uses static variables to remember the current state of the clock and enable lines. These are used for edge detection. In each parse() call, it examines these variables to see the previous state of the lines and compares them to the current state to detect transitions. Before exiting, it updates the clk_was and en_was variables to provide context for the next call. This is a typical way of maintaining simple context information.

This works in simple parsers, but in more complex parsers (like protocol parsers), you probably need to save more than line level context; you will need to save your protocol context (like 'I'm waiting for the sub-command to command 5'). For example, it might receive the first byte of a frame and interpret it as a command. It can probably send back a field at this point to print the command, but it must remember which command was received so that when the next byte is received, it will know what to do with it. No doubt its meaning varies with which command was sent.

State machines are very well suited to this task. A single state variable maintains your current protocol context. Additional static variables are used to remember significant information (like the

edge detection mentioned above.) For example, you might be in state 'waiting for address low in command all-call'. In this case, you probably stored address-high in a previous state. When both are gathered, you might send out a single field with the combined value and then move on to state 'waiting for checksum'. You update your state variable before returning from each event. At the start of each event, you look at the state variable for context in evaluating the current data.

We are not going to discuss state machine design, but there is a lot of information on the web (search for 'finite state machine design'.) We use a simple state machine in the 'RawDAC8045' example.

3.4 Timestamps and TimeScale usage

Timestamps:

DigiView hardware uses very large timestamp counters (> 50 bits). All time is measured relative to the TRIGGER POINT. Time prior to the trigger is represented with negative numbers. Time following the trigger is represented with positive numbers. All timestamp values passed between the application and the plug-in are represented with SIGNED INT64s.

All timestamps sent to your plug-in are guaranteed to be in chronological order. Any given timestamp will be larger than any previously received timestamp. We require that your plug-in send us chronological data as well. For convenience, we allow a few exceptions where your plug-in can send us back-to-back fields with the SAME timestamp.

TimeScale

DigiView uses scaled timestamps in its internal data structures to eliminate the need to deal with floating point values. This greatly improves parsing, displaying and searching performance. For example, a 400MHz sample rate results in a 2.5ns resolution. When we store these timestamps, we scale the time to a whole number by multiplying it by 2. In this case, TimeScale would be 2, telling your plug-in that all timestamps (to and from your plug-in) are scaled 2x. This approach allows the entire application (including your plug-ins) to work with 64 bit integer time.

Many plug-ins do not care about absolute time. The fields generated by the plug-in usually use the timestamp from a particular event. The FIELD timestamp is blindly set equal to the EVENT timestamp; no need to scale it. In these cases, you can ignore the fact the timestamps have been scaled.

The only time a plug-in cares about absolute time is if it is doing timing analysis or an ASYNC

type protocol. In those cases, the plug-in has to be concerned about real-time and must compensate for the scaled values it receives and must return. You might be tempted to convert each received timestamp to real-time by dividing it by the TimeScale. Then you could directly subtract timestamps to measure real-time duration. Then, when you need to send a field back, you would take the real-time timestamp and multiply it by the TimeScale to return properly scaled time. DON'T! This results in a lot of needless floating point math and can have a considerable performance impact.

Instead of converting scaled-time timestamps to real-time, you should convert your real-time parameters to scaled-time. This is a single integer operation that occurs once before the data streaming starts. Then during the parse calls, you continue working with scaled numbers. Many field timestamps will be set to some timestamp received from an event (no math required). Anywhere you require calculated times, you can use integer math to calculate a scaled time. This converts all of the math in the parse-time routines to integers. It also confines the usage of any math at all to the time checks themselves (rather than every received event and every sent field).

Examples:
- If you have a timeout configuration item, then you would multiply it by the TimeScale before storing it for internal use. To check timestamps for the timeout condition:
if ((newscaledtimestamp-oldscaledtimestamp) > scaledtimeout) ///// timed out

- If you have a BAUD RATE parameter, you would immediately convert it to a scaled time duration: ScaledBitTime = (1/baudrate)*TimeScale.

TimeScale usage is demonstrated in the AsyncWD example.

3.5 Future Compatibility

To make your plug-in as compatible as possible with future framework releases, you should:

- Return empty strings for any GetStrList call you do not understand.
- Fully decode the ID and SUBID in SetCfgItem and SetInitItem calls.
- Do not modify the CppCmdParser.cpp file

3.6 Data masks, Pack and FindChannelLimits

The user is free to assign any channel(s) he wants to a signal. They do not have to be consecutive. In fact multi-bit signals (like buses) do not even have to be contiguous. For example, a 4 bit bus could be assigned to channels 3,9,21 and 32. The only rule is that the bits are ordered by their channel numbers. The lowest channel number assigned is the LSB of the bus. One signal's channels can be interspersed with other signal's channels. Mini plug-ins do not have to worry about this as the pre-processor normalizes the data before sending it in an event. It does this by packing the defined bits together and then shifting them to bit 0. In this example you would receive a 4 bit bus using bits 0-3 of the data field in the event.

Full and Hybrid plug-ins can define channel-select options of their own to allow the user to assign additional channels to the plug-in. When you use these, you will receive RAW DATA events whenever ANY of your assigned channels transition. Your plug-in must then extract the bits of interest and normalize them for your own use. The framework provides 2 routines to help with this:

uint64 pack(uint64 dat, uint64 mask, uint64 HighBit, uint64 LowBit)

This does the data extraction, bit packing and shifting needed to normalize a given signal's data. You pass the signal's data mask (returned from the channelselect configuration option) and the current raw data sample. It returns the signal's bits, packed together and 0-bit justified. The remaining parameters are the highest and lowest set bits in the data mask. These should be pre-calculated from the data mask, ONCE during initialization. Since pack is called hundreds of thousands of times pre signal per capture, performance is improved by pre-calculating these limits and then having the pack routine limit its work to this range.

void FindChannelLimits(uint64 mask, uint64 &HighestBit, uint64 &LowestBit)

This is an optimization helper. During configuration, you can use this to pre-calculate the highest and lowest bit positions used in a data mask. These are then passed to the pack routine each time you need to extract a given signal's data. You pass the datamask (from a channel select option) and references to the HighestBit and LowestBit variables.

3.7 Fields

A Field is the smallest unit of information your plug-in can display. It might be derived from a single bit (ACK/NAK, Rd/Wt), multiple bits or even multiple bytes/symbols/characters. For example, you might emit a field called 'SYNC' whenever you see 10 or more 0x55 in a row or an extended quiet period. It's up to you. A field starts at the indicated timestamp and extends until the next field is received.

3.8 Zero Length Fields

Zero Length Fields are normal fields except their start and end times are identical. Normally, we display a field such that it stretches from the time the field began to the point it completed (last bit for example). ASYNC characters have very deterministic start (middle of start bit) and end (middle of end bit) times. Sync fields do not have ending times. For SYNC signals, we usually show the field as stretching from the field's first bit to its last bit, implying that all of these bits make up the field. But how do you display a one bit field where the first bit IS the last bit? This is a zero length field. We chose to show the field as starting at the given bit time and stretching just long enough to allow us to print the field's value and then terminate it. Its closing point is NOT tied to a timestamp. We are labeling a point in time; not a timespan.

Another use for zero length fields is in state signals. Sometimes we like to think of states like a receiving latch sees them; when the state clock transitions, the latch updates and holds the state value. In this case, we simply start a new field each time the clock transitions. Each state is displayed from its starting time until the next field starts. Another way to view states is that we want to see the state value AT the clock edge but don't want to imply that it holds until another strobe. In this case, we could use zero length fields to label each transition with its value at that instant. We would make StartField() call at the clock transition time and then an EndField() call with the same timestamp. Several of our build-in parsers and the examples demonstrate this with the 'show idle' options.

3.9 Frames

A Frame is a grouping of fields. Not all protocols or other parsers will generate frames. In some cases, there is no inherent framing of the data; it is just a stream of data. For example, the output from an A/D converter is a series of measurements. There is no 'first' measurement or other grouping. In this case, you might choose to frame them into chunks of 16 readings each for better readability when displayed in tables. But in general, there is no real framing. For this example, a plug-in could generate all fields with StartField() calls. In another example, assume this mythical A/D converter converted 4 channels of data, each in turn. In this case, there is a natural framing of 4 readings in each group. Your plug-in would send the first reading with a StartFrame() call and the next 3 readings with StartField() calls. Then it could optionally generate an EndFrame() call to terminate the frame if you wanted to see IDLE time before the next frame. Each field could use a different field format to name the fields CH1,CH2,CH3 and CH4. This would enable you to use searches to find things like: 'Find a frame in which CH3 < 0x59 and CH4 > 0x55'. Framing affects waveform display formatting, table list formatting and search capabilities.

3.10 Control Fields: Soft Triggers and Filtering

In addition to the various display fields, the plug-in can send back control fields. Controls fields allow the plug-in to HALT an autorun sequence and/or control whether the current capture should be saved to disk.

Each capture can be saved to disk. Plug-ins and auto-searches can control whether a particular capture will be saved or not. This allows them to act as filters to ensure interesting captures are preserved for later inspection or that uninteresting ones are excluded. You might use these controls instead of HALT controls to capture multiple soft-triggers. There are several settings in the acquisition settings to control the maximum number of captures saved to disk, the amount of disk space to use or preserve and whether we save round-robin or halt when a limit is hit. All of these settings are honored during any capture save.

Whether or not a specific capture is saved to disk is controlled by the following logic:

- If any plug-in or auto-search object issues a FORCESAVE command, then SAVE
- else if any plug-in or auto-search object issues a VETOSAVE request then do NOT SAVE
- else if the default setting (acquisitions options) is set to SAVE, then SAVE
- else do NOT SAVE.

The save/veto controls operate independently of the HALT command and independently of whether we are doing a single capture or running in auto-run mode. You can halt and save for example. Refer to AsyncWD.cpp for an example.

AUTORUNHALT

During auto-run sequences, the software will arm, capture and transfer data continuously until it receives a HALT command from a plug-in or auto-search or the user presses the halt button. This allows plug-ins and auto-searches to act as soft triggers. They can look for conditions that can not be detected with the hardware trigger circuits (like protocol level events or duration measurements beyond the hardware timer limits) and halt the acquisition so that the interesting event is visible on the screen.

It is important to understand that although soft triggers can 'trigger' on very high level, complicated events, they differ from hardware triggers in that they are not guaranteed to capture one-time/infrequent events. They capture a buffer of data and analyze it. If the buffer does not contain the trigger condition, another buffer is captured and analyzed until the trigger condition is found. This means that the target activity that occurs between fetches is never seen. However, if the trigger condition is repetitive, you will eventually catch it. The auto-run sequence combined with an auto-search or a plug-in using the HALT control code, allows you to start DigiView capturing and walk away. If it ever captures your soft trigger condition, it will halt on that capture so that it is on the screen when you return.

FORCESAVE

A FORCESAVE command from any plug-in or auto-search over-rides all VETOes and the default setting and saves the current capture to disk.

VETOSAVE

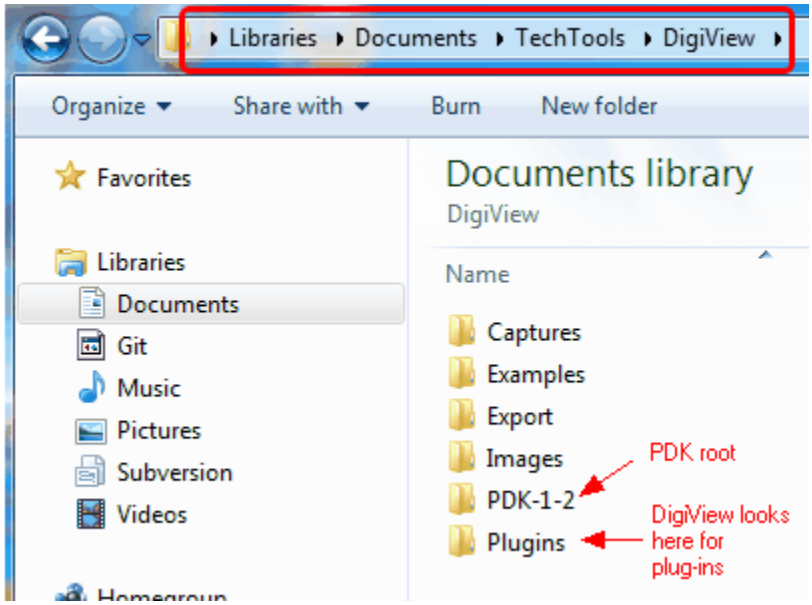
A VETO command from a plug-in or auto-search requests that the current capture NOT be saved. This over rides any default save setting. However, it is over-ridden by any FORCESAVE command from any plug-in or auto-search.

3.11 Finding Your Documents Folder

All of our per-user files are placed under the user's 'Documents' folder. We refer to this as the <Documents> directory or folder. The exact name and file system location of the 'Documents' folder varies under different versions of Windows. You can navigate to it by:

- double-clicking on the 'My Documents' icon on the desktop
- selecting 'Documents' from the START menu
- clicking on 'libraries->Documents' in file explorer.

The default PDK installation and DigiVIew's plugin directory are located at:
<Documents>\TechTools\DigiView\

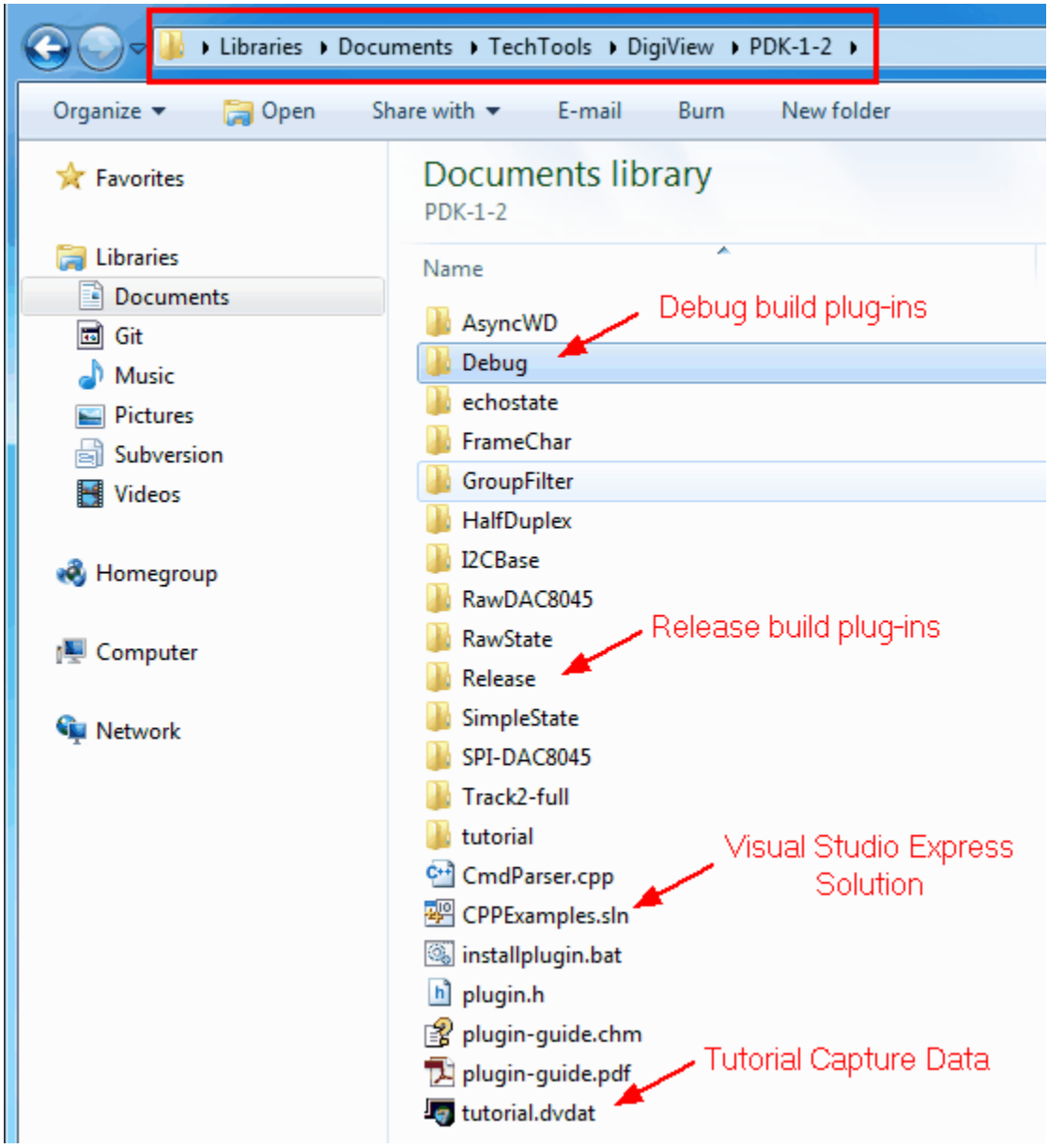


NOTE: Do not place anything except your plug-in's executable (and optionally its help file) in the plug-in directory. EVERYTHING (except *.rtf files) found in this directory is added to our plug-in list. We can not determine if it really is a plug-in until we try to load it at signal creation time. For example, a *.doc file is probably associated with WORD. We would add it to the plug-in list. If you create a signal using this file, we will attempt to launch the file (which would launch WORD) every time we capture data. Likewise, your plug-in source would probably launch your IDE or compiler.

3.12 PDK Directory Layout

If you accepted the default install path during installation, the PDK was installed under your <Documents> directory. The <Documents> folder name and location varies between Windows versions.

See Finding your Documents Folder for help finding <Documents>.



Each example project and its source is placed in its own sub directory under the PDK root.

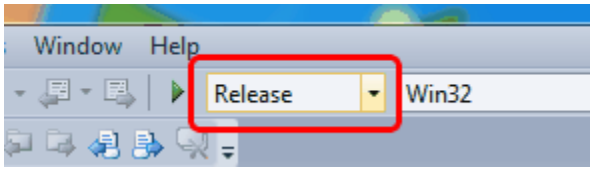
We created a 'solution' (CPPExamples.sln) containing all of the example projects. It is located in the PDK root directory. The examples all use the same CmdParser.cpp and plug-in.h files. These are also placed in the PDK root directory.

Each plug-in project includes the ../cmdparser.cpp and ../plug-in.h files as well as the example specific source file. NOTE: the CmdParser.cpp file handles all of the interaction with the main

application. We included its source as a reference for porting to a different language. There is no need for you to modify it. All of your code goes in the project specific file.

3.13 Final Build

When development is complete, change the project configuration from 'Debug' to 'Release' and do a final build. The resulting files will be smaller and faster. The release versions will be placed under '<PDK root>/release' instead of '<PDK root>/debug'. If your project is set up to 'Auto-Install' the plug-in, then it will also be placed in the DigiView plugin directory.



3.14 Documenting Your Plugin

You can document your plug-in's configuration options and behavior by placing a TEXT or RICH-TEXT (rtf) formatted file in the plug-in directory. Give the file the same root name as your plug-in and an .rtf extension (myplug-in.exe uses myplug-in.rtf). Use the .rtf extension even if the file is plain-old-text. You can use Wordpad or OpenOffice to create RTF files. Of course, notepad can create text files. The built-in viewer does not support advanced features like embedded graphics. Stick to stylized text.

When the user is editing the signal's configuration options, they can press the help button to view your document. This is a good place to remind the user (or yourself) what the plug-in does, what the options do, or anything they should do or avoid ('be sure to select rising edge clock' or 'we ignore SYNC settings for now').

3.15 Runtime DLLs

The development machine will have all DLLs needed for the debug and the release versions you produce. If you move the plug-ins to a machine without the Visual Studio tools installed, it will not have the debug DLLs and might not have the run-time DLLs. You can fetch the run-time DLLs from the Microsoft download site:
http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=5555

Debugging

Part

4 Debugging

Debugging a plug-in is a little different than most applications because it involves 2 interactive processes; DigiView and the plug-in itself. You can not just set breakpoints and then 'run' the plug-in. You have to wait until DigiView starts the plug-in as a separate process. Once the plug-in is running, you can attach a debugger to it and set breakpoints. However, at this point, the entire plug-in runs to completion before you can attach the debugger. You usually want to set breakpoints **before** the plug-in runs but you can't attach the debugger until **after** it runs.

Another complication is the fact DigiView guards communications with the plug-in with a 2 seconds timeout. If the plug-in stops accepting or returning data for more than 2 seconds, DigiView unloads the plug-in and disables any signals using it. Of course, every time your plug-in hits a breakpoint or you pause too long on a single-step, it will timeout.

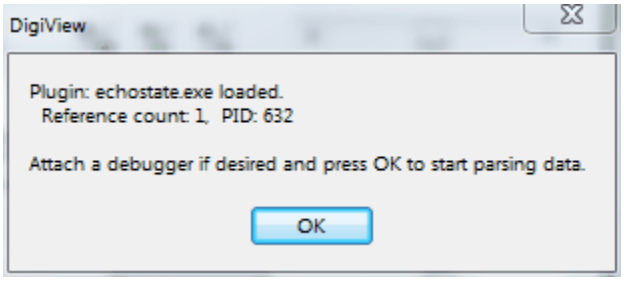
DigiView includes a feature to help with both of these problems.

Debug Plug-ins option

DigiView includes an option setting to assist plug-in debugging. Setting **'Config->Environment->Debug Plug-ins'** causes DigiView to modify its behavior as follows:

Pauses after loading a plug-in

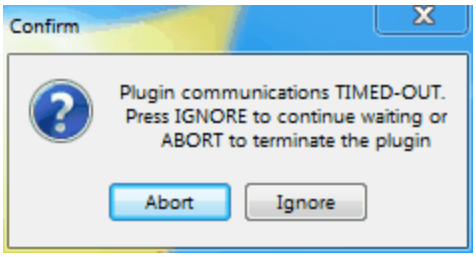
A dialog is presented immediately after loading a plug-in and starting to communicate with it. At this point, the OnLoad routine is the only user code in the plug-in that has executed. This pause gives you a chance to attach a debugger and to set breakpoints in your plug-in code before any other code is executed. Once you select 'OK', the plug-in is interrogated, configured and called to parse the existing capture data.



Traps plug-in communication timeouts

All interaction with a plug-in is guarded with a 2 second watchdog timeout. Normally, if the plug-in does not respond to a command or absorb enough data to allow new commands in

that amount of time, an error is generated. the plug-in is unloaded, and any signals using the plug-in are disabled. When debugging support is enabled, a timeout dialog is presented rather than unloading the plug-in.



If you select 'Ignore', the timer is reset and the operation is retried. This allows you to set breakpoints and single-step your code without DigiView killing your process. It also means that once you select 'Ignore', DigiView will pick up where it left off rather than starting over. Selecting 'Abort' will unload the plug-in and disable associated signals.

The following sections describe how to set up for debugging and some work-flows to deal with these challenges.

4.1 Debug Setup

Need Data But Not Hardware

You will need sample capture data to exercise the plug-in but do not necessarily need DigiView hardware. If you are using one of the example captures, you should unplug your DigiView or select NO if prompted to convert the file to match your hardware. This reduces complications if the DigiView used to capture the data had more channels than your DigiView.

Disable ALL plug-in based signals

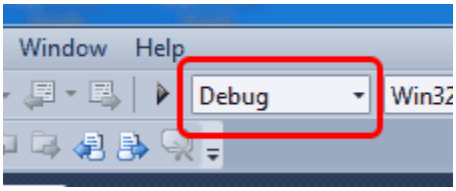
When multiple signals use a single plug-in, DigiView loads a single instance of the plug-in into memory. DigiView will not unload the plug-in until ALL signals using it have been deleted or disabled. It can be annoying to have to disable several signals each time you update the plug-in. Also, each signal will attempt to use the plug-in making it difficult to track exactly which configuration is being debugged. For the smoothest, most consistent debug session, it is usually better to have only one signal enabled at a time that uses your plug-in.

When in debugging mode, DigiView will pause before **each** plug-in is loaded, giving you a chance to attach a debugger to that specific plug-in. Debugging is simpler if there are no

signals using plug-ins except the one you are debugging. Delete or disable all plug-in based signals.

If you already have a signal defined that uses your plug-in, disable it for now, even if you intend to use it for debugging.

Change Solution Configurations to Debug in Visual Studio Express



Turn on Expert settings in Visual Studio Express

Select '**Tools->Settings->Expert Settings**' to enable additional debugging options

See the next section for work-flow suggestions.

4.2 Debug Work-flow

First Load (without using a debugger)

1. We won't be using a debugger for this part so do NOT set Debug Plug-ins Option in DigiView.
2. Create/enable signal
   - click 'Config -> Signals' in DigiView
   - Select your plug-in in the signal type list
   - Click 'Add'.
   - The plug-in will be loaded, enabled, and executed.
3. If you get error messages (other than timeouts):
   - The plug-in will be unloaded
   - The message will describe problems with the configuration string syntax.
   - Make corrections, rebuild, and re-enable the signal until it loads without errors.
4. If you get TIMEOUT messages:

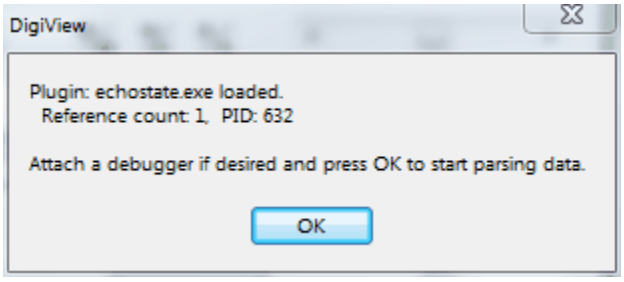
- The plug-in stopped responding and will be unloaded.
- See Using A Debugger below.

5. If the plug-in loads without error but doesn't parse correctly:
   - The plug-in will remain loaded and active.
   - Disable the signal to force unloading of the plug-in.
   - See Using A Debugger below.

Using a Debugger

NOTE: you can not use F5 or 'Debug->Start Debugging' with a plug-in. You must invoke the plug-in with DigiView and then 'attach' the debugger to plug-in after it is running.

1. Set Debug Plug-ins Option in DigiView
2. Enable the Signal in DigiView.
3. DigiView will load the plug-in and execute its onload(). The 'attach debugger' dialog will appear.



4. In Visual Studio, select **'Debug->Attach to Process'**. If you don't see this option, you forgot to select **'Tools->Settings->Expert Settings**' during Debug Setup.
5. Select your **plug-in's name** from the list and click **'Attach'**
6. Open the plug-in source and set breakpoints. See Plug-in Data-flow for information on where you might want to set a breakpoint. You will probably put one at the start of parse() so you can look at variables to determine if the configuration variables were interpreted properly and then start debugging your actual parse() algorithms.

7. Dismiss the 'attach debugger' dialog. DigiView will start interrogating, configuring and streaming data to the plug-in until it hits a breakpoint or completes.

8. If the plug-in hits a breakpoint, DigiView will display a timeout dialog after 2 seconds. You can ignore this and continue single-stepping, viewing variables etc. until you run out of data. (see Streaming and Buffering). When the plug-in runs out of data or fills its output FIFO, it will hang in a loop in CmdParser(). Dismissing the timeout dialog will allow DigiView to absorb your plug-in's data and fill your input FIFO with more data to process. The timeout dialog will re-appear in about 2 seconds. These FIFOs are large so you could debug hundreds of samples before having to dismiss the dialog.

9. When you finish debugging, RUN to completion and dismiss timeout dialog. If DigiView appears to lock-up/freeze at this point, see the notes about Hidden Dialogs.

10. If changes are needed:

    - Disable the signal & close its signal editor
    - Edit and rebuild the plug-in
    - Copy your plug-in to <DigiView's plugin directory> if not using Auto-Install
    - Return to step 2 and repeat as needed.

11. If you run to completion and wish to breakpoint or step the entire process again (without rebuilding):

    - Disable the signal in DigiView. This causes DigiView to unload the plug-in.
    - Re-Enable the signal. This causes DigiView to reload the plug-in and run a full parse cycle.

12. If you just need to debug the parse() portion of the code again (without rebuilding):

    - Open the signal editor and change one of the configuration items. This will skip the loading/unloading and calls to GetStrLits(). Each configuration change does a complete parse sequence.

4.3 Debugging Tips

Task Manager

If Windows refuses to let you replace your plug-in with a new version with a message about being in use, it means that DigiView is still using it or it is a Zombie (abandoned.) This really should not happen but if something goes very wrong with the plug-in and it stops responding to the DigiView application, DigiView tries to kill the process. If Windows can not kill it for some reason, it stays in memory and and the disk copy is locked.

If this happens, first make sure that every signal using your plug-in is disabled. If that is not the problem, then open the Windows task manager (usually available through cntrl-alt-delete or by right-clicking on the task bar) and review the Processes list. Find your plug-in's name in the list, click on it and select 'End Process'. If it is listed multiple times, end all of them. Now you should be able to copy over the new version of the plug-in.

Hidden Dialogs

DigiView presents modal dialogs to allow you to attach a debugger to a plug-in and to report timeouts. These are configured to always be on top of the DigiView application. However, in some circumstances, Windows will ignore the setting and paint the application itself on top of the dialogs. Since the dialogs are modal, Windows will freeze the main application itself (DigiView) until the dialog is dismissed. Since the dialog is hidden behind the application, you can not dismiss it.

Note that this is not a DigiView specific problem. It just seems more confusing when you have 2 interactive programs (DigiView and the Debugger) running at the same time. This seems to happen in Windows if you have a modal dialog open in one application, then switch to another application. When you return to the first application, Windows sometimes paints it on top of its modal dialog, creating this problem.

We have seen this behavior if the timeout dialog is present and you click RUN in the Visual Studio Express debugger to allow the plug-in to complete. When the plug-in finishes, focus is returned to DigiView but it paints in the wrong order.

There are 3 wwork-arounds to this problem:

- Click on DigiView's icon in the task bar TWICE. The first click will sometimes minimize the application or other times appear to do nothing. The second click will usually paint the application and dialog in the correct order so the dialog is on top where you can dismiss it.

- Or dismiss the dialog BEFORE pressing RUN in Visual Studio (assuming you can do so in 2 seconds).
- Or move the dialog to a blank spot on the desktop when it pops up so that you will always have access to it, no matter what order Windows paints items..

Streaming and Buffering

The communications between the DigiView application and your plug-in use overlapping, streaming data packets with FIFO buffering in both directions. Once you hit a breakpoint in your code, and DigiView displays a timeout dialog, you could have several hundred received events queued up for processing. Likewise, you could send several hundred field definitions back to DigiView before filling up the queue. The implications are that once you hit a breakpoint, you could process a lot of data before having to dismiss the timeout dialog.

Eventually you will run out of events to process or you will fill up your TX queue and the plug-in will hang in the CmdParser portion of the template, attempting to communicate with the DigiView app. If this happens, simply dismiss the timeout window so that DigiView can process its end of the data and the plug-in will continue. If you were single stepping or you hit another breakpoint, DigiView will timeout and display the dialog again.

Searches and Triggers

Search and trigger configurations depend on the configuration options from your plug-in. Any changes to the configuration items or the field formats in your plug-in could invalidate the trigger and/or search settings.

Common errors

Configuration string syntax

Fortunately, DigiView does extensive error checking of all of the configuration strings when the plug-in is first loaded. Any errors are reported and the plug-in is unloaded. Most error messages point to the specific field within the specific string with the error. Debugging this portion is usually pretty easy can be done without a debugger. Refer to Configuration Editors for the correct syntax.

Field Chronology

Once the plug-in fully loads, the most common problem is getting fields out of sequence. If your plug-in ever sends back a field with an older timestamp than the previous field, an error is reported

and the plug-in is disabled; no time-travel allowed. However, you can generate back-to-back fields with the SAME timestamp in some circumstances. The following sequences are allowed to have the same timestamp:

- StartFrame or StartField -> EndFrame or EndField (zero length field)
- EndField -> EndFrame (EndFrame over-rides)
- EndFrame -> EndField (EndFrame over-rides)
- EndField -> EndField (2nd one ignored)
- EndFrame -> EndFrame (2nd one ignored)

Unexpected formatting

All formatting is controlled by your SendField calls and the Field Formats you specify. If you add or delete field format specifications to the string list, it will throw off any references to them. Using enums (as opposed to using strl.push_back() calls and hard coded indexes) as demonstrated in the examples goes a long way toward eliminating these mistakes. As a bonus, it makes the code more readable and maintainable. However, using enums and direct indexing makes it easy to miss/skip an entry in the stringlist. These empty strings get converted to: '<empty>'. Generally, the application will complain about 'entry x has too few parameters: <empty>'

If some of your Lookup Table values are not printing, it could be due to a reference to an undefined table or an undefined index into a table. Of course, it could also be due to specifying the same color for the background and the font :)

Ignoring data.bytes[7]

Data.bytes[7] in the parse() calls holds a code that describes the type of event we are receiving. 0x90 means RAW DATA event and 0x80 means parser data event. We generally ignore the value of data.bytes[7] in the examples. This is OK because the framework guarantees that mini plug-ins will never receive raw data events and full plug-ins will never receive parser data events. Constantly checking would be a waste of time. Only hybrid plug-ins receive both types of events and need to differentiate between them.

If you take a mini or full plug-in example and convert it to a hybrid, then forgetting to qualify on byte 7 will cause total confusion. See the 'HalfDuplex' example to see how a hybrid plug-in handles the event type code.

Logging

DigiView currently does not provide any type of logging facility for plug-in debug. However, your plug-in has full access to the PC so it could create log files of its own. If you want the log to cover the lifetime of the plug-in, you could open a log file in the OnLoad() call and ensure it is closed in the OnUnload() routine. If you want it to log a single capture parse, you could open/close it in the StartOfData() and EndOfData() routines. Keep in mind that any logging from within the parser routine could generate a lot of data and could have an impact on DigiView's performance. A low-impact form of logging is to store significant events in memory and then dump them to a file on EndOfData(). For example, you might store the last dozen events and fields in memory. When parsing is complete (or fails), you could dump them, along with some of your current state (bit-number, field count...) to disk.

Performance and stability

Keep in mind that your plug-in becomes a part of the DigiView application. Its performance and stability affects the entire application. At any given time, there can be dozens of plug-ins operating. Also, each plug-in is called to parse the raw data each time new data is captured. The plug-in's parse() routine is very performance sensitive. It can receive hundreds of thousands of calls per data capture, per signal.

DigiView is fairly tolerant to plug-in lockups or crashes. The plug-in is loaded as a separate process. All communications with it use separate threads, timeouts (about 2 seconds) and large FIFOs. If the plug-in stops responding, DigiView will attempt to kill the process. However, killing a process is not always successful and certainly undesirable. If you are having problems with plug-in lock ups, it could affect the stability of the DigiView application or the system.

Plug-in Framework

Part V

5 Plug-in Framework

We provide a template that handles the communications protocol, provides access routines and stubs out routines for your code. The framework invokes several calls in your code to status and configure the plug-in and to parse the captured data into protocol frames and fields. Your code then uses access routines in the framework to send back control and data field descriptions.

A plug-in is written as a console mode program. This makes it very language independent, lightweight on resources and easy to write. There are no DLLs, sockets, pipe handles, byte orders, Windows APIs, etc to deal with and every language supports console I/O. It can be a compiled executable or a script.

We provide a module (CmdParser.cpp) to handle the I/O itself and to handle our communications protocol. Your plug-in code focuses on interpreting the data and generating formatting instructions.

CmdParser.cpp provides main() and takes control when the plug-in is loaded. It interacts with the DigiView application, interprets its commands and forwards specific commands and data to routines in your plug-in code. Your plug-in then uses CmdParser.cpp supplied access routines to return field information to the DigiView application. The section Getting Started above discussed the provided example files, project layout and instruction on how to build the plug-ins.

5.1 Source Files

A complete plug-in consists of 3 files:

plugin.h

CmdParser.cpp

<yourplugincode>.cpp

plugin.h (provided)

This include file defines a few globals and prototypes the access routines in the framework and the stubs in your code. It also defines a union called Data64. We use int64s extensively. This union allows us to access an int64 as a pair of int32s or an array of 8 bytes, as well as an int64.

CmdParser.cpp (provided)

This module handles all communications with the DigiView application. It parses the commands and data and forwards them to your plug-in code as needed. It also provides data output routines your plug-in uses to send back information. We provide the source to this module for your reference, in case you want to port it to another language. You do not need to make any changes to CmdParser.cpp. All of your code goes in the plug-in specific file <yourplug-incode.cpp>.

Your plug-in uses the following routines to interact with the application.

Data Output Routines

Control Routines

Utility Routines

Data Output Routines

These routines allow your plug-in to return field information to the application. They control all framing and formatting of the data.

Parameters

The access routines all share the following parameters:

TIMESTAMP:

This is the time at which the field starts or stops. Note that timestamps must never be less than the previous timestamp. They are allowed to match the previous timestamp in special cases. See the section 'Debug Tips/Common Errors/Field Chronology' and 'Development hints/Zero Length Fields' below.

DATA:

You can supply up to 48 bits of data. This data is interpreted by the Field Format for formatting and display. Usually, the data is simply displayed as a single number. However, the Field Format can extract slices (bit ranges) of the data to display multiple numbers in a field. It can also specify slices to be used as indexes into lookup tables to print data driven text strings. How the bits are used is defined entirely by the Field Formats you specify (see Field Format Syntax below.)

FormatID:

FormatID is an index into the field format list you supplied in the GetStr call. See 'FIELD FORMATS' in 'GetStrLists' for details.

CntlCode:

This is a constant (defined in plug-in.h) defining which control code you wish to send

CNTLHALT

Halts the current Autorun sequence, making the current capture data the final capture.

CNTLSAVE

Forces this capture to be saved to disk (even if vetoed by other signals or autosearches.)

CNTLNOSAVE
Vetoes saving this capture to disk.

void StartField(int64 timestamp, Data64 data, unsigned char FormatID)

This starts a new field at 'timestamp'. It auto-terminates any previous field.
'Data' is interpreted and formatted per the field format specified by 'FormatID'

void StartFrame(int64 timestamp, Data64 data, unsigned char FormatID)

This starts a new field and tags it as a start of frame. It auto-terminates any previous field or frame.

'Data' is interpreted and formatted per the field format specified by 'FormatID'

void EndField(int64 timestamp)

This marks the end of a field. Timestamp is the ending time. This is optional. Fields are auto-terminated by the next StartField or StartFrame. The only reason to use this is if you want to see the field terminated earlier.

void EndFrame(int64 timestamp)

This marks the end of a frame. Timestamp is the ending time. This is optional. Frames are auto-terminated by the next StartFrame. The only reason to use this is if you want to see the frame terminated earlier.

Control Routines

void SendControl(int64 timestamp,unsigned char CntlCode)

Sends the specified control code to the application. 'Timestamp' is currently ignored but might be used in future versions to log the time of the control code.
See 'Development Tips->Control Fields' for usage information.

Utility Routines

void FindChannelLimits(uint64 mask, uint64 &HighestBit, uint64 &LowestBit)

A utility routine your plug-in can use to help work with raw data efficiently. See 'Data Masks, pack and findchannellimits' below

uint64 pack(uint64 dat, uint64 mask, uint64 HighBit, uint64 LowBit)

A utility routine your plug-in can use to help work with raw data efficiently. See 'Data Masks, pack and findchannellimits' below

<yourplugincode.cpp>

This is where all of your code goes. It should '#include' or link with the CmdParser.cpp file and supply the functionality for the following 8 routines: They all must be defined but most of them can simple return. Few plug-ins will need to use all of them. (See 'Development Tips->Plug-in Dataflow' for details on when and why each are called.)

void OnLoad()

Called when the plug-in is first loaded. If multiple signals use the same plug-in, it is loaded only once. This is used for global initialization, memory allocation, etc. You might use this for any memory allocation that lasts for the plug-in's lifetime.

void GetStrList(int ID, vector<string> &strl)

Called multiple times each time a signal is created, enabled or its editor is opened. ID specifies which string list is needed. These routines simply fill in 'strl' with the requested set of strings. To allow maximum compatibility with future DigiView releases, your plug-in should return an empty list when it receives an ID it does not understand. There are currently 6 string sets defined:

ID 0: Return the plug-in description.

These lines are displayed in the signal editor to describe this plug-in's name, purpose, copyright, etc. For best results, keep to 4 or fewer lines.

ID 1: Return configuration options

When the user creates a signal based on your plug-in, they are presented with a signal editor dialog to allow them to configure your plug-in. These strings describe the configuration options and their parameters. If the plug-in is a mini-plug-in, these items are added to the pre-parser's items and this section might be minimal or even empty. If this plug-in is a full-plug-in (based on RAW data), this section will include every option needed to extract and interpret the data (which channels are being used and for what purpose, the BAUD rate, clock polarity...)

Each configuration option LABEL in your plug-in should be unique. Also, if you are using one of the built-in pre-processors, your labels should not conflict with its labels. The labels are displayed to the user and are also used for internal reference.

(see: Configuration Editors for Available editors and their configuration syntax)

ID 2: Field Formats

Called multiple times each time a signal is created, enabled or edited. These describe how to format and display fields. As you generate fields in PARSE, you tag each field with an index into this list to describe to the application how to display and format the field. You must provide at least one field format description.

The general format is: Name,Background Color, Font Color, Display Format as described below:

Name:

Any text that describes this format. This shows up in the TABLE views (when field names are enabled) and in the search dialogs. Names should be unique.

Background Color:

Color used to fill the field background in the waveform view and the table cells. If left blank, the default signal background color is used. You can use one of the predefined colors listed below or specify a standard RGB triplet (808080 = middle gray).

Font Color:

Color used for the field text. If left blank, the default signal font color is used. You can use one of the predefined colors listed below or specify a standard RGB triplet (0000FF = RED).

Display Format:

Describes what to print in a given field. Can contain a mixture of text and data slices. A 'Data Slice' defines a range of bits from the provided data. The full format looks like: TEXT {T%H:L} TEXT {T%H:L}....

The text portions can contain any printable character except '{','}' or ',' (we will probably remove the ',' restriction by full release)

'TEXT' is an optional text string.

It can contain any printable character except '{' or '}'

'{T%H:L}' is an optional data slice. Each part is option.

'T%'

signifies that this slice selects a string from a lookup table (returned from GetStrList(5,lst) where 'T' is the table number. If T is omitted, then table 0 is used. If 'T%' is omitted, then this is a DATA slice rather than a lookup slice. The slice defines an int32 rather than a lookup string.

'H:L'

defines a range of bits to extract from the data. H is the highest bit and L is the lowest bit. If L is omitted, it defaults to the lowest bit (0). IF H is omitted, it defaults to the highest bit possible for this slice type (L+31 for data or L+13 for lookups, but limited to 47).

The largest number that we can display is 32bits long. Any single DATA slice with a range exceeding 32 bits will produce an error.

Lookup table indexes are limited to 14 bits. Any single LOOKUP slice with a range exceeding 14 bits will produce an error.

Some DATA slice examples:

{3:1} = the number defined by bits 3->1

{3:} = {3:0}

{:12} = {43:12}

{:40} = {47:40} (limited by highest bit available)

{3} = {3:0}

{} = {31:0}

Some Lookup table slice examples:

{2%47:40} = the string in lookup table 2 at the offset
specified by bits 47->40 (byte 5 of the data)

{%43:40} = {0%43:40}

{1%3} = {1%3:0}

{4%:30} = {4%43:30}

{3%:40} = {3%47:40} (limited by highest bit available)

{1%3} = {1%3:0}

{1%} = {1%13:0}

Pre-defined colors

The following (non case-sensitive) color constants can be used in the Background and Font color parameters:

'black','brown','red','orange','yellow','green','blue','violet','gray' and 'white'

ID 3: Pre-Processor Name

Tells the application which pre-processor your plug-in needs. Most plug-ins will use one of the built-in parsers as a pre-processor to handle low level protocol issues (like bit extraction or clock polarity). Others will need full control. Plug-ins are responsible for all framing and formatting so the built-in parser options dealing with those issues are removed when used a a pre-processor.

See 'Pre-Processors' below for details

ID 4: Framework Version

Tells the application the minimum Framework version this plug-in needs. For now, there is only one version so return '1'

ID 5: Lookup tables

Optional list of Lookup table (substitution) entries. If you are not using lookup tables, return an empty list.
Each entry defines a single lookup/substitution string.
The general format is: table number,index,substitution string as described below:

Table Number:
A number between 0 and 31

Index:
a number between 0 and 4095 relative to the current table

Substitution String:
A string of printable characters. Can be any printable character except ',' (this restriction will probably be removed by full release)

For optimal performance and resource usage:
The table numbers should be consecutive and start at 0. Likewise, the indexes for each table should be consecutive and start at 0.
e.g.:
"0,0,ACK"
"0,1,NACK"
"1,0,RD"
"1,1,WT"
"1,2,ERROR"

void SetInitItem(unsigned char ID, unsigned char subID, int value)

Called multiple times immediately before each parse run to set some global parameters

ID 0: Set Timescale

DigiView uses scaled timestamps in its internal data structures. This call provides the scaling

factor so you can convert between scaled-time and real-time if needed. Many plug-ins do not need to worry about this. See the section TIMESTAMPS & TIMESCALE in the development hints section for details.

ID 1: First Timestamp

Tells the plug-in the timestamp of the very first raw sample. This int64 is sent in 2 calls with subid 0 = LSB and subid 1 = MSB. The first and final timestamps are useful for timing analysis.

ID 2: Final timestamp

Tells the plug-in the timestamp of the very last raw data. This int64 is sent in 2 calls with subid 0 = LSB and subid 1 = MSB. The first and final timestamps are useful for timing analysis.

void SetCfgItem(unsigned char ID, unsigned char subID, int value)

Called after all SetInitItem calls and before the StartData call. Called multiple times before each parse run to set the user selected parameters for this signal. 'ID' identifies which cfg items is being set. It is an index into the configurations strings you provided in the GetStrList call (the first item listed is ID=0.) VALUE is a single INT32. Each type of configuration object defines what the return values mean. If the object requires more than a single INT32 to define it settings, your plug-in will receive multiple calls, with increasing subIDs. See the section on Configuration Option Syntax for the return values from each option. See the SIMPLESTATE example plug-in for typical handling.

void StartOfData()

Called after all cfg items are set and before the data starts streaming. Gives you a chance to examine the cfg items after ALL of them have been set, make adjustments and any initializations.

void Parse(int64 timestamp,Data64 rawdata)

This is the heart of the parser. DigiView will stream EVENTS (packets of time/data information) to your plug-in through this routine. Parse() examines the data and stream back FIELDS (packets of time/field information) through the Data Output Routines defined above. Note this gets called on the order of 250,000 times per signal per capture so efficiency is important.

void EndOfData()

Called at the end of each parse run for a given signal. Gives you a chance to flush any final field information and do any clean up.

void OnUnload()

Called just before the plug-in is removed from memory. If multiple signals are using the plug-in, it will be called ONCE, when the final signal is disabled or deleted. You might use this to free any resources allocated in OnLoad(). DO NOT SEND any fields back from this call. The EndOfData call is your last chance to do that.

5.2 Pre-Processors

The built-in interpreters can be used as pre-processors for your plug-in.

Every built-in interpreter consists of 2 parts; a **pre-processor** and a **post-processor**.

Pre-Processor

The pre-processor handles the link level protocol extraction. This includes things like detecting clock edges, honoring enables, shifting bits and detecting protocol defined start/stop or error conditions.

Post-Processor

The post-processor is responsible for formatting and framing the data (when framing is not part of the protocol). It determines field colors and how the data is printed in each field. Mini-plug-ins REPLACE the internal post-processor. The output (events) from the internal pre-processor is routed to your plug-in. Your plug-in's output (fields) are stored in the signal's internal state table. Full plug-ins accept raw data events and generate fields directly. In effect, they replace both the pre-processor and the post processor.

Available Pre-Processors:

ASYNC (Asynchronous) | SYNC (Synchronous) | SPI | I^2C | STATE | I^2S | CAN | RAW

ASYNC

This is sometimes (incorrectly) referred to as RS-232 or generally as a 'SERIAL' port. It is characterized by a BAUD rate setting and a lack of a clock signal. Symbols (Bytes or Characters) of data are sent as a start bit (0), followed by a pre-configured number of data bits, an optional parity bit and a stop bit (1). Bits are blindly sampled at the BAUD rate. The symbol is 'framed' by a 0 start bit and a 1 stop bit. Note that this use of the term 'framing' refers to framing a single symbol and is different than our use as a protocol frame.

A protocol frame would consist of one or more of these symbols. There is no defined protocol framing at the pre-processor level. In essence, this pre-processor is acting like a UART and your plug-in acts like the software/firmware that gathers data from the UART and interprets it, possibly into higher levels of framed data.

Configuration Options provided by the pre-processor

Data

Selects which physical channel to assign to the DATA bus

Baud Rate:

Selects from a list of standard BAUD rates or 'use custom'

Custom Baud (bits/sec):

The BAUD rate to use if BAUD RATE is set to 'use custom'

Data Bits

Selects the number of data bits in a character

Parity/9bit Address flag

Selects from odd,even,one,zero,non standard parity settings.

Also allows selection of 9bit addressing mode with and address

field flagged with a '1' or with a '0'

Glitch Filter (% of bit)

Select noise filter setting of none-10% of a bit width

Sync (skip transitions)

Specifies how many transitions to ignore at the start of the buffer.
useful for syncing up when capture starts mid-character

MSb First:

Specifies that bits are received in MSB first order (VERY rare)

Events

This pre-processor uses event flags to indicate which events occurred. The data event occurs at the middle of the start bit time and also includes any parity or framing error flags. Additional parity and/or framing events occur later at their respective times. In the built-in post processor, we handle the data event first (ignoring any additional parity or framing errors) and then handle any error events as they occur, allowing us to show each as a separate field at their respective timestamps. Setting the flags during the data event allows your plug-in to decide whether to display the corrupt data or not.

Event Format:

```
byte[0] = data (during data event..else ignore)
byte[6] = event flags:  (Break,End,Parity,Frame,X,X,Address,Data)
```

Data

Indicates that byte[0] holds a complete data byte. The timestamp marks the middle of the start bit time. NOTE: any parity or framing errors associated with this byte are flagged as well. They can be ignored if desired because any such errors will be reported later as independent, timestamped events at their respective bit times.

Address

Indicates the parity bit position matched the user defined '9-bit address mode' level and that byte[0] holds the gathered address byte. The timestamp marks the middle of the start bit time. NOTE: any framing error associated with this byte is flagged as well. It can be ignored if desired because any such error will be reported later as an independent, timestamped event at the STOP bit time.

Framing Error

Indicates the middle of the STOP bit position was low. The timestamp is the middle of the stop bit time.
NOTE this reference to FRAMING has nothing to do with our use of the word as defined in the TERMINOLOGY section above. This is referring to timing framing of the character. When

you receive these framing errors, it means that the baud rate or one of the other low level parameters is set wrong, or the transmitter and receiver (us) are out of sync.

Parity Error

Indicates the parity calculation did not match the user's selection. The timestamp is the middle of the parity bit time.

End

Indicates the timestamp of the end of the character. Typically used to send an ENDFIELD type field back.

Break

Indicates that the line was held low for greater than a character time

SYNC

Synchronous is not really a protocol but a concept. At its core, it refers to serial data, strobed in by a separate clock signal.

- The data is sampled on one or both of the clock edges.
- There are no predefined number of bits per field (symbol) or fields per frame.
- There are no predefined framing indicators. The field lengths usually vary within a frame and are often data dependent.

There are many link-level implementations of serial protocols and many higher levels of protocols built on top of them.

The pre-processor honors the select signal (ignores clocks while disabled/deselected) but ignores the FrameSYNC and Field signals. If the user enables either of these, the pre-processor simply detects transitions on the selected lines and reports them to the plug-in.

Configuration Options provided by the pre-processor

Clock

Selects which physical channel to assign to the CLOCK

Data

Selects which physical channel to assign to the DATA bus

Select

Selects which physical channel to assign to the ENABLE.
The enable can be disabled if not used

Frame SYNC

Selects which physical channel to assign to the FRAME SYNC.
This can be used to identify frame limits
The FRAME SYNC can be disabled if not used.

Field SYNC

Selects which physical channel to assign to the FIELD SYNC.
This can be used to identify field limits

The FIELD SYNC can be disabled if not used.

Clock On

Selects which edge of the clock to use to strobe in data

Select Level

Selects the active level for the Select signal

Events

This pre-processor can generate more than 1 event at a time. It sets 1 or more event flags in data byte[6] to indicate which events occurred at this timestamp. It also updates some status bits in that same byte to indicate the current state of some of the control signals. The event flags and status levels are defined below:

DATAEVENT FLAG (bit 7 : 0x80)

When this bit is set, state data was strobed in at this time. The data field holds the clocked data.

SELECTEVENT FLAG (bit 6 : 0x40)

When this is set, the SELECT channel transitioned. The SELECTSTATE tells us if it went active or inactive

FrameSYNCEVENT FLAG (bit 5 : 0x20)

When this is set, the FrameSYNC channel transitioned. The Frame SYNCLEVEL tells us the new level

Field SYNCEVENT FLAG (bit 4 : 0x10)

When this is set, the FieldSYNC channel transitioned. The Field SYNCLEVEL tells us the new level

SELECTSTATE (bit 2 : 0x04)

The current state of the select signal. 1 => enabled, 0=> disabled (regardless of the logic level on the physical channel)

Frame SYNCLEVEL (bit 1 : 0x02)

The current logic level of the FrameSYNC channel. The preparser assumes nothing about the meaning of this signal so it passes the actual logic level to you

Field SYNCLEVEL (bit 0 : 0x01)

The current logic level of the FieldSYNC channel. The preparser assumes nothing about the meaning of this signal so it passes the actual logic level to you

SPI

SPI is a specific synchronous protocol. It uses separate data-in and data-out lines and a common clock. Select is optional. SPI is very common in microcontrollers due to the simplicity of the hardware implementation. This pre-processor handles most common variations (including 2 phase clocking).

To avoid the ambiguity of choosing a master vs. slave viewpoint, the data lines are often labeled: MISO (Master-In-Slave-Out) and MOSI (Master-Out-Slave-In). The select signal is called Slave Select (SS). Since the data streams use a common clock, enable and field length, their fields share a common timestamp. The pre-processor sends the timestamp and both fields (MOSI and MISO) in each Data Event.

Configuration Options provided by the pre-parser

Clock Channel

Selects which physical channel to assign to the CLOCK

MOSI Channel

Selects which physical channel to assign to the MOSI data

MISO Channel

Selects which physical channel to assign to the MISO data

SS Channel

Selects which physical channel to assign to SS (slave select)

Clock MOSI On

Specifies which clock edge to use to strobe in MOSI data

Clock MISO On

Specifies which clock edge to use to strobe in MISO data

SS active level

Specifies the active level for the SS (slave select) signal

Field Idle Timeout (0 to disable)

A new field is started if no new bits are seen for more than the specified time.
Set to 0 to disable.

Skip Bits (to sync)

Specifies how many bits to ignore at the start of the buffer.
Useful for syncing up when capture starts mid-field

Field Length (bits)

Specifies the data field length from 4 to 24 bits.

Events

Event format:

bytes[2-0] = MISO data
bytes[5-3] = MOSI data
bytes[6] = event flags

The pre-processor uses event flags to indicate which events occurred. Multiple flags (in limited combinations) can be set at the same time.

Data flag: 0x80

Data fields contain a data capture. Can be accompanied by a Partial flag and/or a SSEN flag. Will never occur with an End or SSDIS flag.

Partial flag: 0x40

Indicates the captured data is incomplete. It was interrupted before gathering the full specified number of bits (usually by SS going inactive or a timeout). This flag never occurs without a Data flag; It is a modifier to the Data Flag. Your plug-in can decide whether to tag these differently, ignore them or treat them like normal data.

SSEN flag: 0x20

SSEN flag indicates the Slave Select (SS) signal transitioned to the enabled state. SSEN transitions can occur alone or with a Data flag.

SSDIS flag: 0x10

SSDIS flag indicates the Slave Select (SS) signal transitioned to the disabled state. SSDIS transitions can occur alone or with an End flag.

End flag: 0x08

Marks the end time of a field. The data fields are meaningless. Can occur alone or with a SSDIS flag.

I2C

Extracts all I2C events including START,STOP,ACK,NAK. Also sends separate events for special codes, address, R/W and data fields. Note that framing is inherent in this protocol so the preparser sends START and STOP events to your plug-in.

Configuration Options provided by the pre-processor

Clock(SCL)

Selects which physical channel to assign to the CLOCK

Data(SDA)

Selects which physical channel to assign to the DATA

Glitch Filter

Selects the amount of noise filtering. Should be set to 50ns for low speed operation and reduced for faster speeds

Skip Bits (to sync partial frame)

Specifies how many bits to ignore at the start of the buffer.
Useful for syncing up when capture starts mid-frame

Decode Addr 000-0001-d as

Selects between the standard I2C decoding for this address range or decoding it as normal 7 bit devices.

Decode Addr 000-001X-d as

Selects between the standard I2C decoding for this address range or decoding it as normal 7 bit devices.

Decode Addr 111-11XX-d as

Selects between the standard I2C decoding for this address range or decoding it as normal 7 bit devices.

Decode HS Master Codes as

Selects between the standard I2C decoding for this address range or decoding it as normal 7 bit devices.

Decode 10bit Codes as

Selects between the standard I2C decoding for this address range or decoding it as normal 7 bit devices.

Truncated fields

Specified whether to show truncated/partial fields or not. 1 bit truncated fields common and unavoidable so the options include showing only if > 1 bit.

Events

The built-in I2C preparser generates 16 events. A single event is sent at a time. The event code is placed in byte[6] and is encoded as shown below:

START (0)

Generated whenever the start condition is detected on the bus.

START-BYTE (1)

Generated when the first byte holds the special code: 0000 0001. Under normal operation, the plug-in should expect to receive a NAK event followed by a Repeated start (Sr) event and then any normal ADDRESS event.

ADDRESS (2)

Generated for all general 7 bit addresses in the first byte. byte[0] contains the entire 8 bit value of the first field. This includes the 7 bit address in the upper 7 bits and the direction bit in the LSB. One would normally ignore the direction bit and just grab the address at this point. A DIR event will follow shortly to timestamp the direction bit. It will include the same data[0] as this call, allowing you to handle the address and/or direction in either/both calls.

GENERAL-CALL (3)

Generated when the GENERAL-CALL code (0000 0000) is detected in the first byte. Under normal operation, the plug-in would then expect to receive an ACK event followed by the general-call sub code in a DATA event.

CBUS (4)

Generated when the first byte contains the CBUS code (0000 001X). Following this event, the preparser ignores all bus activity until a STOP condition is detected. The plug-in will receive a STOP event when the CBUS activity completes.

HSMASTER (5)

Generated when the special HSMASTER code (0000 1XXX) is detected in the first byte. The XXX is the master's code. Data[0] contains the full 8 bit code. Under normal operation the plug-in would expect this to be followed by a NAK event, a repeated start event and then a normal 7bit address event. High speed operations remains in effect until a STOP event is received.

RESERVED (6)

Generated when an address within the 2 reserved ranges is detected in the first byte. data[0] contains the address and direction bit. A DIR event will follow with direction bit's timestamp and the same data.

10BITADDR (7)

Generated when the special 10Bit Code (1111 0XX) is detected in the upper 7 bits of the first byte. Notice the XX bits are the upper 2 bits of a 10 bit address. The remaining 8 bits come from the next byte or are assumed from the context. The pre-parser does not look at the direction bit, the next byte and/or the presence of a Repeated start to decode the actual 10 bit address. It simply reports the detection of this code. Of course, additional events will follow to report DIR, data and start/repeated starts so that a plug-in could determine the full 10 bit address just as a slave device would.

DIR (8)

Generated for the Direction bit. The LSB of data[0] indicates the bit value; 1=> Read, 0=> Write. The upper 7 bits of data[0] contain the 7bit address this DIR event refers to. It can be ignored as it was sent to the ADDRESS event earlier.

ACK/NAK (9)

Generated for the Ack/Nak bit. The LSB of data[0] indicates the bit value; 1=> NAK, 0=> ACK

DATA (10)

Generated at the start of each byte of data in the payload. data[0] contains the data.

STOP (11)

Generated whenever the stop condition is detected on the bus.

Truncated (12)

Generated when a partial byte of data was received.

RESTART (13)

Generated whenever the start condition is detected on the bus WITHOUT a preceding STOP condition. This is called a Repeated Start.

FIELD-IDLE (14)

Generated to timestamp the end of a byte of data. Usually used to allow display of idle periods between the last data bit and the ACK.NAK bit.

STATE

Sends the data captured at each clock edge as an event. Also send SELECT and SYNC events (if defined). The pre-processor honors the select signal (ignores clocks while disabled/deselected) but ignores the SYNC signal. If the user enables the SYNC channels, the pre-processor simply detects transitions on that line and reports them to the plug-in.

Configuration Options provided by the pre-processor

Clock Channel

Selects which physical channel to assign to the CLOCK

Data Channels

Selects which physical channels to assign to the DATA bus

Enable Channel

Selects which physical channel to assign to the ENABLE.

The enable can be disabled if not used

Frame SYNC Channel

Selects which physical channel to assign to the FRAME SYNC.

This can be used to identify frame limits

The FRAME SYNC can be disabled if not used.

Clock On

Selects which edge of the clock to use for strobing in data

Enable Level

Selects the active level for the Enable signal

Events

This preparser can generate more than 1 event at a time. It sets 1 or more event flags in data byte[6] to indicate which events occurred at this timestamp. It also updates some status bits in that same byte to indicate the current state of some of the control signals. The event flags and status levels are defined below:

DATAEVENT FLAG (bit 7 : 0x80)

When this bit is set, state data was strobed in at this time. The data field holds the clocked data.

SELECTEVENT FLAG (bit 6 : 0x40)

When this is set, the SELECT channel transitioned. The SELECTLEVEL tells us if it went active or inactive

SYNCEVENT FLAG (bit 5 : 0x20)

When this is set, the SYNC channel transitioned. The SYNCLEVEL tells us the new level

SELECTLEVEL (bit 2 : 0x04)

The current state of the select signal. 1 => enabled, 0=> disabled (regardless of the logic level on the physical channel)

SYNCLEVEL (bit 1 : 0x02)

The current logic level of the SYNC channel. The preparser assumes nothing about the meaning of this signal so it passes the actual logic level to you

I2S

The I2S built-in pre-parser is available beginning with DigiView Version 8.1.

The I2S decoder will pre-parse the data based on the configured options below. Since this protocol does not have the typical framing of multiple fields your plug-in will only receive two event types (Left Channel, Right Channel). The data for each event, the type of event and the starting of the next event is based on the Word Length and Word Select options.

Configuration Options provided by the pre-processor

Data Channel

Selects the channel to use for DATA.

Clock Channel

Selects the channel to use as the CLOCK.

WS Channel

Selects the channel to use as the Word Select (WS). The word select determines whether the data is for the Left or Right audio channel.

Convert Data to Unsigned

Selects whether the preprocessor should treat the data as signed and convert it to unsigned when showing values or plotting.

Word Length

Selects the bit width for the word length from 4 bits to 32 bits. This setting determines the length of each field.

Events

Event Format:

Byte[7] = Event Type
Bytes[3:0] = Value - (32 bits)

Left Channel (Byte[7] = 0):

Bytes[3:0] hold the Left Audio Channel data. Use StartField() to format.

Right Channel (Byte[7] = 1):

Bytes[3:0] hold the Right Audio Channel data. Use StartField() to format.

CAN Bus

The CAN BUS built-in pre-parser is available beginning with DigiView Version 8.1.

Extracts all CAN events including Active Error Frames, Overload Frames, Bit Stuffing Errors and Form Errors. Also sends separate events for each field including control bits, delimiters, Extended ID and CRC. Note that framing is inherent in this protocol so the preparser sends the SOF event at the beginning of a frame and will send the End or IDLE event to terminate the frame.

Configuration Options provided by the pre-processor

Data Channel

Selects which physical channel to capture and decode.

Bit Rate/Duration & Scale:

Specify a value for the Bit Rate or the Bit Duration, then select a scale for the value entered. Scale selections include Nano seconds (ns), Micro seconds (us), Milli seconds (ms), Baud, KiloBaud (KBaud) and MegaBaud (MBaud). Note that the value must be an integer (no floating point.) To specify something with a decimal point, select the next lowest range and enter a whole number. For example, 115.2 KBaud would be entered as 115200 Baud and 12.31us would be entered as 12310ns.

Sample Point (%bit)

Specifies where we should sample the data within each bit cell as a percentage of the bit width. This defaults to mid-bit (50%) and can be adjusted to account for bus propagation delays, transceiver delays, bandwidth limitations, etc. This roughly corresponds to the combined settings of the SYNC_SEG, PROP_SEG and PHASE_SEG_1 mentioned in the CAN specification. Alternatively, it could be viewed as the total bit width - PHASE_SEG_2.

Sync Jump Width (%bit)

Specifies the percentage of a bit width to allow for resynchronization adjustments. This corresponds to the SJW parameter in the CAN specification.

The CAN specification allows for a wide tolerance on node oscillators. This is accomplished by requiring that nodes resynchronize on passive->dominate edges. The receivers compare the actual timing of these edges with the ideal timing at the specified baud rate and then make adjustments to their internal timers to resynchronize with the incoming data. This parameter specifies the maximum adjustment we will make when resynchronizing.

Increasing this number increases our ability to properly decode packets involving nodes with low

accuracy oscillators at the expense of increased noise sensitivity. Lower numbers improve noise rejection but reduces our ability to work with nodes with low accuracy oscillators. You usually set this lower if all nodes use crystal oscillators for their baud rate reference, and higher if any of them use ceramic resonators or other low accuracy sources for their baud rate references.

Be aware that the (SAMPLE-POINT + SJW) should be less than 100% and that (SAMPLE-POINT - SJW) should be greater than 0.

Filter Glitches <= (%bit)

Specifies filtering of pulses that are less than or equal to the specified percentage of the Bit width. DigiView has a much higher bandwidth than most CAN receivers so it is capable of capturing glitches or noise pulses that normal receivers might not even see. Also, many CAN receivers have different levels of filtering available. This option lets you tell us how relatively good we should be at rejecting noise pulses. A setting of 0 tells us to process full bandwidth data with poor rejection. Increasing this setting makes us simulate higher immunity parts. Selection range is from 0% to 10%.

Events

The built-in CAN preparser generates 25 Event Types which are identified in byte[7]. If an error occurs or an invalid value is detected, byte[6] will be non-zero. Bytes[5:0] can contain up to 48 bits of data. The content of Data will depend on the event.

Event Format:

Byte[7] = **Event Type**

Byte[6] = **Flags -** (see the Event Type tables for flags supported by each event)

Form Error	(bit 0 : 0x01)
Invalid Value	(bit 1 : 0x02)
Bit Stuffing Error	(bit 2 : 0x04)

Bytes[5:0] = **Data -** Contains up to 48 bits of data. Contents will be one of the following based on the event:

1. Field Value - Will contain the value of the field event.
   Applicable Events: Base ID, EXTID, DLC, CRC

2. Bit Count - For multi-bit events that have no relevant data. Will contain the number of successive bits denoting the event. If the number of bits or state (Dominate or Recessive) is incorrect, the Form Error flag is set in Byte[6].
   Applicable Events: EOF, EF-D, OL-D, OVERLOAD, ERROR, END, IDLE, IFS-I, BITSTUFF

3. Bit state - Denotes the state of single bit field events. 0 = Dominate, 1 = Recessive. If the state is wrong for the event type, the Form Error flag is set in Byte[6] for appropriate events.

Applicable Events: SOF, SRR, RTR, IDE, R1, R0, CRC-D, ACK, NAK, ACK-D

Field Event Types - Byte [7] =

Use StartField() to format Field event types. EndField() is not required as any field will automatically terminate when the next StartField(), StartFrame() or EndFrame() is called or when the end of data is reached. If a form error occurs, the Error flag will be set for applicable events.

Byte [7]	Event	Information	Flag Support	Output
0	Unknown Field	reserved		StartField()
1	Base ID	Data[0:1] holds the 11 bit Frame ID.		StartField()
2	SRR	Control bit - Substitute Remote Request	Form Error	StartField()
3	RTR	Control bit - Remote Transmission Request Data[0] = 0, RTR of Data Frame. Data[0] = 1, RTR of Remote Frame.		StartField()
4	IDE	Control Bit - Identifier Extension bit		StartField()
5	R1	Control bit - Reserved 1 (extended frames only)	Form Error	StartField()
6	R0	Control bit - Reserved 0	Form Error	StartField()
7	EXTID	Extended ID Data[2:0] holds the 18 bit Extended ID.		StartField()
8	DLC	Data Length Count On DATA FRAMES, Data[0] should specify the number of data events to follow. A maximum of 8 Data events will be sent. REMOTE FRAMES do not have data events. Data[0] specifies the length of data requested. The CRC event should follow the DLC event in Remote Frames. Current specifications limit DLC to a maximum value of 8, so the Invalid Value flag is set when this value is	Invalid Value	StartField()

Byte [7]	Event	Information	Flag Support	Output
		greater.		
9	Data	Data[0] is a data byte.		StartField()
10	CRC	Cyclic Redundancy Code Data[1:0] holds the CRC sequence.		StartField()
11	CRC-D	Delimiter - CRC	Form Error	StartField()
12	ACK	ACK slot - CRC match Acknowledgement		StartField()
13	NAK	ACK slot - No CRC match Acknowledgement		StartField()
14	ACK-D	Delimiter - ACK slot	Form Error	StartField()
15	EOF	Delimiter - End of Frame Data[5:0] holds Bit Count	Form Error	StartField()
16	EF-D	Delimiter - for Error Frames Data[5:0] holds Bit Count	Form Error	StartField()
17	OL-D	Delimiter - for Overload Frames Data[5:0] holds Bit Count	Form Error	StartField()

Note: Each event is sent in chronological order and already has a unique Timestamp. If adding your own field by splitting events into multiple StartField() calls, do not use a previous or duplicate time stamp or one that is greater than the event's timestamp. Make certain each field's timestamp is (> last) and (<= current).

Frame Event Types - Byte [7] =

Use StartFrame() or EndFrame() accordingly. If an error occurred, the Error flag will be set for applicable events. Events IFS-I and BITSTUFF are always an error.

Byte [7]	Event	Information	Flag Support	Output
18	SOF	Start Of Frame		StartFrame()
19	OVERLOAD	Overload Frame, Data[5:0] holds Bit Count	Form Error	StartFrame()
20	ERROR	Error Frame, Data[5:0] holds Bit Count	Form Error	StartFrame()
21	END	Frame end (for formatting purposes). Sent when IFS (Inter-Frame Space) completed without a Form Error or being interrupted by an Overload Frame.		EndFrame()

Byte [7]	Event	Information	Flag Support	Output
		Data[5:0] holds Bit Count		
22	IDLE	Frame end (for formatting purposes). Sent while resynchronizing and finding a BUS IDLE condition. Data[5:0] holds Bit Count		EndFrame()
23	IFS-I	Inter-Frame Space Interruption (interruptions other than by Overload or Error Frames). Data[5:0] holds Bit Count	Form Error (always set)	StartFrame()
24	BITSTUFF	Bit Stuffing Violation. Data[5:0] holds Bit Count	Bit Stuffing (always set)	StartFrame()

Note: The IDLE event is almost equivalent to the END event. Both events indicate a termination of the frame but under different circumstances. In either event, if you want to see the "idle" period between frames, use the EndFrame() routine to terminate it. This is not a requirement since all frames automatically terminate when a new frame begins or the end of data is reached.

RAW

Sends an event for the very first and last data samples as well as any time any of the plug-in's defined channels transition.

Configuration Options provided by the pre-processor :

NONE. The plug-in takes full responsibility for specifying all needed parameters/options.

Events

Every event is a RAW DATA event. The data parameter contains a snap-shot of the data channels at the timestamp. Your plug-in uses the masks returned from channel-select objects to extract the pieces of data you are interested in.

5.3 Configuration Editors

When the user creates a signal or clicks on the signal name, a signal editor opens. This editor contains a number of configuration editors, allowing the user to configure the signal parser. These editors allow the user to specify which channels are used and for what purpose. They also allow setting things like baud rates, clock edges, field sizes; whatever you want the user to be able to configure.

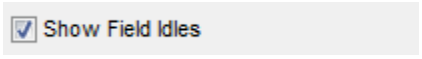
When a plug-in is loaded, the application will generate a number of GetStrList() calls to the plug-in to retrieve information about which editors the plug-in wants to display and their parameters.

Each time the application wishes to refresh the data interpretation, it makes a number of SetCfgItem() calls to the plug-in to inform it about the user's configuration choices.

The following sections document the available configuration editors, their syntax and return values.

Check box

Example: "Show Field Idles,checkbox,true"



GetStrList() syntax: Label,checkbox,default

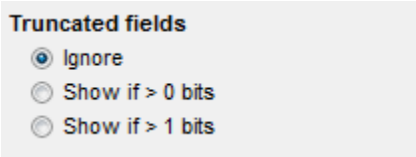
default is the initial state. It can be 0,1,true,false.yes or no

SetCfgItem() values:

subID 0: 1 for checked, 0 for unchecked

Radio group

Example: "Truncated fields,radio,0,Ignore,Show if > 0 bits,Show if > 1 bits"



GetStrList() syntax: Label,radio,default,item0,item1...

default is the item index to select initially.

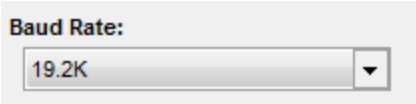
item0,item1... are the options

SetCfgItem() values:

subID 0: 0-based index of selected item

Combo box

Example: "Baud Rate:,combo,2,4800,9600,19.2K,38.4K"



GetStrList() syntax: Label,combo,default,item0,item1...

default is the item index to select initially.

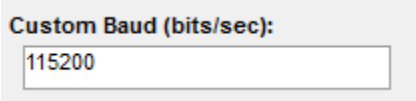
item0,item1... are the options shown in the pull-down

SetCfgItem() values:

subID 0: 0-based index of selected item

Integer Editor

Example: "Custom Baud (bits/sec):,edit,115200"



GetStrList() syntax: Label,edit,default

default: the contents of the edit box. It can be 1 or more comma separated INT32s

SetCfgItem() values:

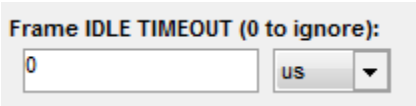
subID 0: the 1st integer in the list

subID 1: the 2nd integer in the list

... for each integer up to 32 total

Time Editor

Example: "Frame IDLE TIMEOUT (0 to ignore):,timeedit,0"



GetStrList() syntax: Label,timeedit,default

default is the initial time in ns

SetCfgItem() values:

subID 0: Lower 32bits of the entered time (in ns)

subID 1: Upper 32bits of the entered time (in ns)

Spinner

Example: "Sync (skip transitions),spinner,3,0,32,1"

GetStrList() syntax: Label, spinner,default,min,max,step

Default: initial value (must be >= MIN and <= Max

Min: minimum value returned

Max: maximum value returned

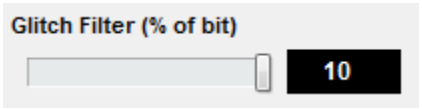
Step: step size (spinner snaps to these increments)

SetCfgItem() values:

subID 0: Spinner position/value

Slider

Example: "Glitch Filter (% of bit),slider,10,0,10,1"



GetStrList() syntax: Label, slider,default,min,max,step

Default: initial value (must be >= MIN and <= Max

Min: minimum value returned

Max: maximum value returned

Step: step size (spinner values increment by this value)

SetCfgItem() values:

subID 0: Slider position/value

Channel Select

Example: "MOSI Channel,2,1,1,true,true"

GetStrList() syntax: Label,chanselect,default value, min,max,showinvert,showdisable

default: a 64bit mask for the default channel selection

min: minimum number of channels the user is allowed to select (> 0)

max: maximum number of channels the user is allowed to select (<= #Channels)

showinvert: 1,true, or yes => show the invert option else hide it

showdisable: 1,true, or yes => show the disable option else hide it

SetCfgItem() values:

subID 0: flags.

Bit 0 is the INVERT status (1 => checked)

Bit 1 is the DISABLE status (1 => checked)

subID 1: Lower Int32 of the selection mask

subID 2: Upper Int32 of the selection mask

Plug-in Examples

Part VI

6 Plug-in Examples

The examples are not production ready code. They are intended to demonstrate how to write plug-ins. As such, they focus on clarity more than completeness. Additionally, some of them were tested with manufactured data. For example, the Track2 plug-in is written from a specification and tested with generic SYNC signals. We did not actually test it against captured credit card swipes. The point is that these examples are focused on demonstrating the mechanics of writing plug-ins. It is very likely they would require additional modifications for actual use but provide a solid, working baseline.

6.1 EchoState

A minimal, yet functional plug-in in 24 lines of code. It is based on the STATE pre-processor. It simply prints each state in YELLOW.

6.2 SimpleState

A mini plug-in based on the STATE pre-processor. Demonstrates adding a few simple user options, performing framing and simple formatting.

6.3 RawState

A full plug-in implementing a basic state parser. Demonstrates parsing raw data events, edge detection, and use of FindChannelLimits and pack().

6.4 I2CBase

A mini plug-in based on the I2C pre-processor. It is an exact replacement for the internal post-processor. It demonstrates using multiple field formats, lookup tables, framing and zero-length fields. This is a good starting point for implementing higher level protocols or project specific substitutions (addr 0x5 = 'D/A' or 'U3'...).

6.5 FrameChar

A mini plug-in based on the ASYNC pre-processor. Starts a new frame whenever a specific character is received. Uses a specific escape character to allow the start-of-frame character to appear in the payload.

6.6 HalfDuplex

A hybrid plug-in based on the ASYNC pre-processor. It specifies a new signal (Direction) to watch. The Direction line determines which end of the bus is sending. The plug-in modifies the field formatting to indicate which end of the link sent the data. It also starts a new frame each time the bus changes directions.

6.7 AsyncWD

A mini plug-in based on the ASYNC pre-processor. In addition to formatting and printing each ASYNC character, it looks for excessive bus dead time. If the time between characters exceeds the user specified value, it forces a save of this capture, and/or halts any auto-run sequence. Demonstrates use of TimeScale, calculating timing, and use of control fields. Also shows inserting non-data related fields into the data display; very useful for auto-searches.

6.8 Track2-full

A full plug-in to decode track 2 from magnetic strip cards (like credit cards). Demonstrates channel extraction, edge detect, using channel invert option, and parity calculation.

6.9 SPI-DAC8045

A mini-plug-in based on the SPI pre-processor. This customizes the SPI parser to decode the data sent to a Nation Semiconductor DAC8045S085. Demonstrates use of multiple data slices and lookup tables to do in-place data decoding. It is less pretty than a full decoder, but is still very functional and easy.

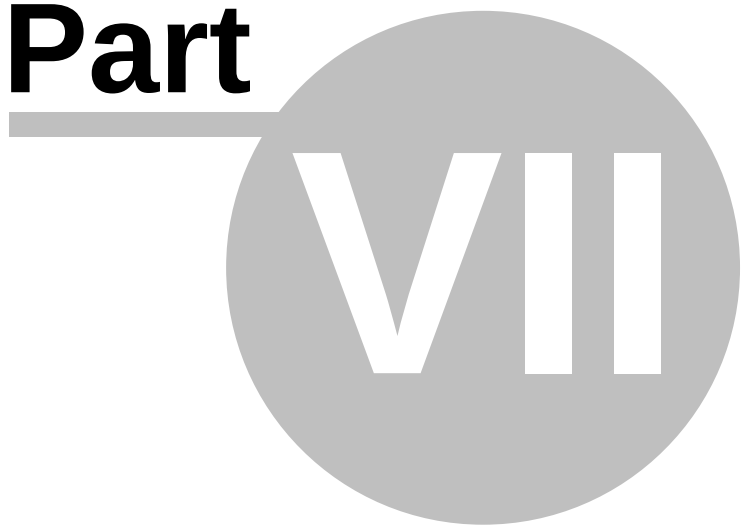
6.10 RawDAC8045

A full plug-in to parse the Nation Semiconductor DAC8045S085. Demonstrates edge detection, framing, idle fields, lookup tables and maintaining context through static vars and a state-machine.

6.11 GroupFilter

A full plug-in to demonstrate glitch filtering across a group of signals.

Disclaimers and Restrictions



7 Disclaimers and Restrictions

Use of this Plug-in Developer's Kit and the sample source provided constitute acceptance of the following disclaimers and restrictions:

7.1 No Warranties

This software is provided 'As Is', without any express or implied warranty of any kind, including but not limited to any warranties of merchantability, noninfringement, or fitness of a particular purpose. TechTools does not warrant or assume responsibility for the accuracy or completeness of any information contained within this software.

7.2 Limits on Liability

In no event shall TechTools be liable for any damages (including, without limitation, lost profits, business interruption, or lost information) rising out of use of or inability to use this software, even if advised of the possibility of such damages. In no event will TechTools be liable for loss of data or for indirect, special, incidental, consequential (including lost profit), or other damages based in contract, tort or otherwise. TechTools shall have no liability with respect to the content of the software or any part thereof, including but not limited to errors or omissions contained therein, libel, infringements of rights of publicity, privacy, trademark rights, business interruption, personal injury, loss of privacy, moral rights or the disclosure of confidential information.

7.3 Use and Redistribution

You may use the files included in this Plug-in Developers Kit to develop DigiView plug-ins for your own use. You may also distribute your derived works as long as the copyrights, disclaimers and restrictions are retained in the source files and followed. Any other use is prohibited without express, written permission from TechTools. This covers all files not explicitly documented as 'NOT REDISTRIBUTABLE'. Should a source file not contain a statement listing the disclaimers and allowed usage, the following must be inserted into the file before distribution:

```
This source file is part of the TechTools Plug-in Development Kit.
Copyright (c) 2011 by TechTools

DISCLAIMERS:
 - NO WARRANTIES
    This software is provided 'As Is', without any express or implied
    warranty of any kind, including but not limited to any warranties
    of merchantability, noninfringement, or fitness of a particular
    purpose. The Copyright holders do not warrant or assume responsibility
    for the accuracy or completeness of any information contained within
    this software.
 - LIMITATION OF LIABILITY
    In no event shall the copyright holders be liable for any damages
    (including, without limitation, lost profits, business interruption,
```

or lost information) rising out of use of or inability to use this software, even if advised of the possibility of such damages. In no event will the copyright holders be liable for loss of data or for indirect, special, incidental, consequential (including lost profit), or other damages based in contract, tort or otherwise. The copyright holders shall have no liability with respect to the content of the software or any part thereof, including but not limited to errors or omissions contained therein, libel, infringements of rights of publicity, privacy, trademark rights, business interruption, personal injury, loss of privacy, moral rights or the disclosure of confidential information.

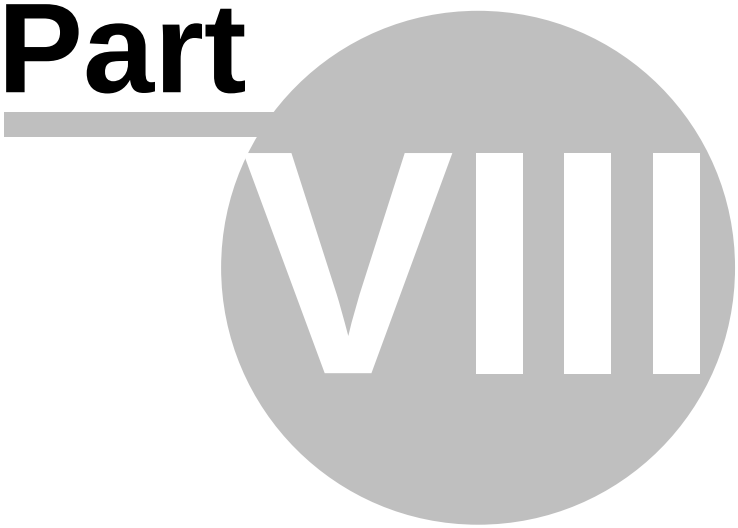
Use and redistribution of this software or of derived works, in source or compiled form is permitted as long as the following restrictions are observed:

- The above copyright(s), disclaimers and these restrictions are retained in the source files and, if distributed in compiled form, must be duplicated in documentation or other materials and provided with the distribution.
- The derived work is used exclusively as a plug-in to the TechTools DigiView software, to process data captured with TechTools hardware.
- The copyright holders' names may not be used to endorse or to promote any product or derived works.

Any other use is prohibited without express, written permission from TechTools.

email: support@tech-tools.com, sales@tech-tools.com
web: www.tech-tools.com Voice:972-272-9392 FAX: 972-494-5814

Contact Information



8 Contact Information

You can contact TechTools at any of the following numbers:

email: support@tech-tools.com, sales@tech-tools.com

web: www.tech-tools.com

Voice: 972-272-9392

FAX: 972-494-5814

Index

- E -

- F -

- G -

- H -

- I -

- L -

- M -

- N -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -

- Y -

- Z -