Plug-in Developer's Guide

General Overview

Plugins 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.

Terminology

Plugins 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.

1. **Samples:**
   The raw data gathered from the hardware at its sample rate.
2. **Channels:**
   These are the physical connections to the target. Our Logic Analyzers have 9,18 or 36 channels.
3. **Active Channels:**
   The physical channels that are assigned to active signals. These are the channels the hardware is monitoring.
4. **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.)
5. **Signal Interpreter**
   This refers to the routines used to translate the raw captured data into the representation in the waveforms and list views. The signal interpreter uses the channel mapping and signal configuration options to extract data from the raw capture data, interpret it and format it for display.
6. **Pre-processor and Post-processor / Pre-parser and Post-parser**
   All signal interpreters 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. We often use processor and parser interchangeably.

7. **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.

8. **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.

9. **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.

10. **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 or field counts or timeouts to mark frame boundaries. Frames' start/end conditions are specially tagged/ formatted fields.

Types of Plugins

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

1. **Mini Plugins**
   Mini plugins 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 plugin can concentrate on higher level issues like formatting, adding another level of protocol, soft triggering or filtering. The plugin depends on the pre-parser supplied user options for the basic protocol configuration. The mini plugin can add additional options if needed (see the I2C plugin 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-plugin based on the internal ASYNC pre-parser. The pre-parser will extract the ASYNC characters for you (like a UART would). Your plugin would inspect the characters and look for your protocol's commands, parameters and any framing indications. Your plugin would then display the protocol as you see fit.

2. **Full Plugins**
   A full plugin 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 plugin is monitoring.

It does not provide any user configurable options. All user options for the protocol (including channel-selects) are specified by the plugin. The plugin 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.

3. **Hybrid Plugins**
   A hybrid plugin is based on an internal pre-parser like the mini-plugin. However, it also specifies additional channels to watch and assumes all responsibility for them. In this configuration, DigiView sends the plugin 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 plugin 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 plugin could monitor the DIRECTION control line and adjust the display formatting to differentiate which end of the link sent the message. The 'HalfDuplex' example plugin demonstrates this.

Capabilities

Plugins can extend the DigiView application in a number of ways:

1. **Modify formatting**
   The 'echostate' example demonstrates a functional plugin in 24 lines of code. It simply displays state fields in a different color. This is the most basic operation a plugin could do; change the way the data looks. A plugin 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.

2. **Add parameters or control signals to an existing protocol**
   Plugins 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 plugin examples add a 'SHOW FIELD IDLE' parameter, controlling whether idle periods should be shown between fields.

3. **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.

4. **Add entirely new protocols**
   Using a full plugin, you could implement new protocols from the link level up. You have full access to everything captured (related to your plugin). 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 plugin. CANbus is different enough from

the built-in parsers that it would have to be built as a full plugin. The plugin 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.

5. **Analyze the data contents and/or timing**

   Plugins 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'). Plugins 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.

6. **Control DigiView's run-time behavior**

   In addition to or instead of printing errors or timing information as field values, the plugin 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 plugin 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.

Plugin 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 plugin 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 plugin 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 plugin code focusses on interpretting the data and generating formatting instructions.

CmdParser.cpp provides main() and takes control when the plugin is loaded. It interacts with the DigiView application, interprets its commands and forwards specific commands and data to routines in your plugin code. Your plugin then uses CmdParser.cpp supplied access routines to return field information to the DigiView application. The section 'Development Tips' below discusses the provided example files, project layout and instruction on how to build the plugins.

A complete plugin consists of 3 files:

1. **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.

2. **CmdParser.cpp (provided)**
   This module handles all communications with the DigiView application. It parses the commands and data and forwards them to your plugin code as needed. It also provides access routines your plugin 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 plugin specific file <yourplugincode.cpp>.

   Your plugin uses the following routines to communicate with the application.

   1. **Access Routines**
      These routines allow your plugin to return field information to the application. They control all framing and formatting of the data.

      1. **Parameters**
         The access routines all share the following parameters:

         1. **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.

         2. **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.)

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

         4. **CntlCode:**
            This is a constant (defined in plugin.h) defining which control code you wish to send

        1. **CNTLHALT**
           Halts the current Autorun sequence, making the current capture data the final capture.
        2. **CNTLSAVE**
           Forces this capture to be saved to disk (even if vetoed by other signals or autosearches.)
        3. **CNTLNOSAVE**
           Vetoes saving this capture to disk.

    2. **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'
    3. **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'
    4. **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.
    5. **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.

2. **Control Routines**
    1. **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.

3. **Utility Routines**
    1. **void FindChannelLimits(uint64 mask, uint64 &HighestBit, uint64 &LowestBit)**
       A utility routine your plugin can use to help work with raw data efficiently. See 'Data Masks, pack and findchannellimits' below
    2. **uint64 pack(uint64 dat, uint64 mask, uint64 HighBit, uint64 LowBit)**
       A utility routine your plugin can use to help work with raw data efficiently. See 'Data Masks, pack and findchannellimits' below

3. **<yourplugincode.cpp> (can be based on one of the example files)**
   This is where all of your code goes. It should include the plugin.h header file and supply the functionality for the following 8 routines: They all must be defined but most of them can simple return. Few plugins will need to use all of them.   (See 'Development Tips->Plugin Dataflow' for details on when and why each are called.)

   1. **void OnLoad()**
      Called when the plugin is first loaded. If multiple signals use the same plugin, 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 plugin's lifetime.

   2. **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 'stl' with the requested set of strings. To allow maximum compatibility with future DigiView releases, your plugin should return an empty list when it receives an ID it does not understand. There are currently 6 string sets defined:

      1. **ID 0: Return the plugin description.**
         These lines are displayed in the signal editor to describe this plugin's name, purpose, copyright, etc. For best results, keep to 4 or fewer lines.

      2. **ID 1: Return configuration options**
         When the user creates a signal based on your plugin, they are presented with a signal editor dialog to allow them to configure your plugin. These strings describe the configuration options and their parameters. If the plug-in is a mini-plugin, these items are added to the pre-parser's items and this section might be minimal or even empty. If this plugin is a full-plugin (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 plugin should be unique. Also, if you are using one of the built-in pre-parsers, your labels should not conflict with its labels. The labels are displayed to the user and are also used for internal reference.

         The following documents each type of option editor available and its syntax:

         1. **Check box: syntax: Label,checkbox,default**
            default is the initial state. It can be 0,1,true,false.yes or no

            return value[0] = 1 for true, 0 for false

         2. **Radio group: syntax: Label,radio,default,item0,item1...**
            default is the item index to select initially.
            item0,item1... are the options

return value[0] = selected item index

3. **Combo box: syntax: Label,combo,default,item0,item1...**

   default is the item index to select initially.
   item0,item1... are the options shown in the pull-down

   return value[0] = selected item index

4. **Integer Editor: syntax: Label,edit,default**

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

   return value[0] = the first integer in the list
   return value[1] = the 2nd integer in the list
   ... for each integer up to 32 total

5. **Time Editor: syntax: Label,timeedit,default**

   default is the initial time in ns

   return value[0] = Lower 32bits of the entered time (in ns)
   return value[1] = Upper 32bits of the entered time (in ns)

6. **Spinner: 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)

   return value[0] = Spinner position/value

7. **Slider: 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)

   return value[0] = Slider position/value

8. **Channel Select: 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
   max: maximum number of channels the user is allowed to select

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

return value[0] = flags.
  Bit 0 is the showinvert setting
  Bit 1 is the show disable setting

return value[1] = Lower Int32 of the selection mask
return value[2] = Upper Int32 of the selection mask

3. **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:

   1. **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.

   2. **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).

   3. **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).

   4. **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)

      1. **'TEXT' is an optional text string.**

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

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

1. **'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.

2. **'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.

3. **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}

4. **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}

5. **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'

4. **ID 3: Preparser Name**
   Tells the application which preparser your plugin needs. Most plugins will use one of the built-in parsers as a preparser to handle low level protocol issues (like bit extraction or clock polarity). Others will need full control. Plugins are responsible for all framing and formatting so the built-in parser options dealing with those issues are removed.
   See 'Pre-Parsers' below for details

5. **ID 4: Framework Version**
   Tells the application the minimum Framework version this plugin needs. For now, there is only one version so return '1'

6. **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:

   1. **Table Number:**
      A number between 0 and 31

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

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

   4. **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"

3. **void SetInitItem(unsigned char ID, unsigned char subID, int value)**
   Called multiple times immediately before each parse run to set some global parameters

   1. **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 plugins do not need to worry about this. See the section TIMESTAMPS & TIMESCALE in the development hints section for details.

   2. **ID 1: First Timestamp**
      Tells the plugin 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.

   3. **ID 2: Final timestamp**
      Tells the plugin 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.

4. **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 plugin 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 plugin for typical handling.

5. **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.

6. **void Parse(int64 timestamp,Data64 rawdata)**
   This is the heart of the parser. DigiView will stream EVENTS (packets of time/data information) to your plugin through this routine. Parse will examine the data and stream back FIELDS (packets of time/field information) through the access routines defined above. Note this gets called on the order of 250,000 times per signal per capture so efficiency is important.

7. **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.

8. **void OnUnload()**
   Called just before the plugin is removed from memory. If multiple signals are using the plugin, 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.

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. 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. The post-parser 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-plugins REPLACE the internal post-processor. The output (events) from the internal pre-parser is routed to your plug-in. Your plug-in's output (fields) are stored in the signal's internal state table. Full plugins accept raw data events and generate fields directly. In effect, they replace both the pre-processor and the post processor.

1. **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 re-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. Each field in 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 plugin acts like the software/firmware that gathers data from the UART and interprets it, possibly into higher levels of framed data.

1. **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)

2. **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)

1. **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.

2. **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.

3. **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.

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

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

6. **Break**
   Indicates that the line was held low for greater than a character time

2. **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 preparser honors the select signal (ignores clocks while disabled/deselected) but ignores the FrameSYNC and Field signals. If the user enables either of these, the preparser simply detects transitions on the selected lines and reports them to the plugin.

1. **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

2. **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:

   1. **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.
   2. **SELECTEVENT FLAG (bit 6 : 0x40)**
      When this is set, the SELECT channel transitioned. The SELECTSTATE tells us if it went active or inactive

3. **FrameSYNCEVENT FLAG (bit 5 : 0x20)**
   When this is set, the FrameSYNC channel transitioned. The Frame SYNCLEVEL tells us the new level
4. **Field SYNCEVENT FLAG (bit 4 : 0x10)**
   When this is set, the FieldSYNC channel transitioned. The Field SYNCLEVEL tells us the new level
5. **SELECTSTATE (bit 2 : 0x04)**
   The current state of the select signal. 1 => enabled, 0=> disabled (regardless of the logic level on the physical channel)
6. **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
7. **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

3. 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 preparser 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-Slavae-Out) and MISO (Master-In-Slave-Out). 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 preparser sends the timestamp and both fields (MOSI and MISO) in each Data Event.

1. 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.

2. **Events**

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

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

1. **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.
2. **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 plugin can decide whether to tag these differently, ignore them or treat them like normal data.
3. **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.
4. **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.
5. **End flag: 0x08**
   Marks the end time of a field. The data fields are meaningless. Can occur alone or with a SSDIS flag.

4. 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 plugin.

1. 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.

2. **Events**

The build 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:

1. **START (0)**
   Generated whenever the start condition is detected on the bus.
2. **START-BYTE (1)**
   Generated when the first byte holds the special code: 0000 0001. Under normal operation, the plugin should expect to receive a NAK event followed by a Repeated start (Sr) event and then any normal ADDRESS event.
3. **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.
4. **GENERAL-CALL (3)**
   Generated when the GENERAL-CALL code (0000 0000) is detected in the first byte. Under normal operation, the plugin would then expect to receive an ACK event followed by the general-call sub code in a DATA event.
5. **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 plugin will receive a STOP event when the CBUS activity completes.
6. **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 plugin 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.
7. **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.
8. **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.

9. **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.

10. **ACK/NAK (9)**
   Generated for the Ack/Nak bit. The LSB of data[0] indicates the bit value; 1=> NAK, 0=> ACK

11. **DATA (10)**
   Generated at the start of each byte of data in the payload. data[0] contains the data.

12. **STOP (11)**
   Generated whenever the stop condition is detected on the bus.

13. **Truncated (12)**
   Generated when a partial byte of data was received.

14. **RESTART (13)**
   Generated whenever the start condition is detected on the bus WITHOUT a preceding STOP condition. This is called a Repeated Start.

15. **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.

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

   1. **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

2. **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:

   1. **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.
   2. **SELECTEVENT FLAG (bit 6 : 0x40)**
      When this is set, the SELECT channel transitioned. The SELECTLEVEL tells us if it went active or inactive
   3. **SYNCEVENT FLAG (bit 5 : 0x20)**
      When this is set, the SYNC channel transitioned. The SYNCLEVEL tells us the new level
   4. **SELECTLEVEL (bit 2 : 0x04)**
      The current state of the select signal. 1 => enabled, 0=> disabled (regardless of the logic level on the physical channel)
   5. **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

6. **RAW**

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

   1. **Configuration Options provided by the pre-processor :**
      NONE. The plugin takes full responsibility for specifying all needed parameters/options.
   2. **Events**
      Every event is a RAW DATA event. The data parameter contains a snap-shot of the data channels at the timestamp. Your plugin uses the masks returned from channel-select objects

to extract the pieces of data you are interested in.

Development Tips

1. **Plugin 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 plugins, stream the data through the plugin before storage.
   Multiple signals can use the same plugin. Therefore, each signal sends its configuration to the plugin before each parse run. The configuration items are defined by the parser and independently selected by the user for each signal.

   1. **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 plugin is loaded into memory if it is not already loaded
      - the plugin's OnLoad() routine is called if it was not already loaded
      - the plugin's GetStrList is called once for each stringlist
      - the user is presented with a signal editor dialog to configure the signal
      - the user's selections for THIS signal are stored

   2. **On New DATA**

      Each time the capture data changes, the following happens:
      - the plugin's SetInitItem is called multiple times to set some globals
      - the plugin's SetCfgItem is called multiple times to configure the plugin with the user's settings for THIS signal
      - the plugin's StartOfData is called to allow final preparations
      - Data events from the preparser are streamed to the plugin's Parse routine. Parse uses calls to SendField to stream back field information
      - When all of the events have been sent, the plugin'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)

   3. **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.

   4. **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.

   5. **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.

6. **On User changing configuration items**

   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.

2. Streaming and Context

It is important to realize that data is streamed to your plugin and it is expected to stream back its data. Your plugin will not have random access to the data or even know how much data it will receive. Since your plugin is sharing the system with a couple dozen other plugins, 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 plugin must process a single piece of data at a time and immediately return. This requires a certain mindset during development.

The plugin 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 parser() 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. Timestamps and TimeScale usage

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 plugin that all timestamps (to and from your plugin) are scaled 2x. This approach allows the entire application (including your plugins) to work with 64 bit integer time.

Many plugins do not care about absolute time. The fields generated by the plugin 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 plugin cares about absolute time is if it is doing timing analysis or an ASYNC type protocol. In those cases, the plugin 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 into 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.

4. Future Compatibility

To make your plugin 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

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

   1. **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.

   2. **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.

6. **Fields**

   The Field is the smallest unit of information your plugin 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.

7. **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 EendField() call with the same timestamp. Several of our build-in parsers and the examples demonstrate this with the 'show idle' options.

8. 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 plugin 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 plugin 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.

9. Control Fields: Soft Triggers and Filtering

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

Each capture can be saved to disk. Plugins 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 plugin or auto-search object issues a FORCESAVE command, then SAVE
- else if any plugin 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.

1. **AUTORUNHALT**

   During auto-run sequences, the software will arm, capture and transfer data continuously until it receives a HALT command from a plugin or auto-search or the user presses the halt button. This allows plugins 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 is does not contain the trigger condition, it fetches another buffer until if finds it. This means that the target activity that occurs between fetches is never seen. However, if the trigger condition repeats, you will eventually catch it. The auto-run sequence combined with an auto-search or a plugin 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.

2. **FORCESAVE**

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

3. **VETOSAVE**

   A VETO command from a plugin 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 plugin or auto-search.

10. Finding the Plugin Directory

All of our per-user files are placed under 'My Documents'. The file system location of 'My Documents' varies under different versions of Windows, but can be accessed by DBL-clicking on the 'My Documents' icon on the desktop or selecting it from the START menu. The plugin directory is in located at: <My Documents>\TechTools\DigiView\Plugins

NOTE: Do not place anything except your plugin's executable (and optionally its help file) in the plugin directory. EVERYTHING with an executable extension or a file association (except *.rtf files) found in this directory is added to out plugin list. We can not determine if it is a plugin 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 plugin 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 plugin source would probably launch your compiler.

11. Examples

We provide a number of plugin examples to get you started. You can develop plugins with the language/compiler of your choice. All examples were developed 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 plugins without additional expense. It also ensures that you start with known, functional examples. You can download the Visual C++ Express or Visual Studio Express 2010 from:
http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express

The example source and project files are provided as a ZIP file and can be found under the DigiView install directory. You should unzip this file into a working directory in your user files area (NOT under the DigiView install directory and NOT under <My Documents>/TechTools/Plugins).

1. Examples structure

We created a 'solution' (CPPExamples.sln) containing all of the example projects. This is the file you should launch from the examples root directory. Each example project and its source is placed in a sub directory. The examples all use the same CmdParser.cpp and plugin.h files. These are placed in the examples root directory. The plugin projects include the ../cmdparser.cpp and ../plugin.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.

2. First Build

When you first launch Visual Studio, press F7 to create executables for all of the examples. The results will be found in the <examples root>/Debug directory (NOT in each project's debug directory).

3. Descriptions

The examples are not production ready code. They are intended to demonstrate how to write plugins. As such, they focus on clarity more than completeness. Additionally, some of them were tested with manufactured data. For example, the Track2 plugin 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 plugins. It is very likely they would require additional modifications for actual use but provide a solid, working baseline.

1. **Echostate**
   A minimal, yet functional plugin in 24 lines of code. It is based on the STATE pre-parser. It simply prints each state in YELLOW.
2. **SimpleState**
   A mini plugin based on the STATE pre-parser. Demonstrates adding a few simple user options, performing framing and simple formatting.
3. **RawState**
   A full plugin implementing a basic state parser. Demonstrates parsing raw data events, edge detection, and use of FindChannelLimits and pack().
4. **I2CBase**
   A mini plugin based on the I2C pre-parser. 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'...).
5. **FrameChar**
   A mini plugin 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. **HalfDuplex**
   A hybrid plugin 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 plugin 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.
7. **AsyncWD**
   A mini plugin 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.
8. **Track2-full**
   A full plugin to decode track 2 from magnetic strip cards (like credit cards). Demonstrates channel extraction, edge detect, using channel invert option, parity calculation.

9. **SPI-DAC8045**

   A mini-plugin based on the SPI pre-parser. 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.

10. **RawDAC8045**

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

4. Creating your own project

To add your own projects to this solution:

1. Select View->Solution Explorer (if it is not visible)
2. Right-click on the first line (Solution 'CPPExamples....)
3. Select Add New Project
4. Select Win32 Console Application, enter a project name and press OK
5. Select NEXT and select Empty Project then finish
6. Right-click on the resulting project (in solutions explorer)
7. Select 'Add new item'
8. Select 'C++ file' and give it a name (same as project OK)
9. Right-click on the project again and Select 'Add existing item'
10. navigate up 1 directory and select CmdParser.cpp and plugin.h
11. Copy one of the examples files into your new file to use as a base line.
12. Do a test build and you should have a functional plugin
13. Modify to suit.

5. 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 'release' instead of 'debug'.

6. Runtime DLLs

The development machine will have all DLLs needed for the debug and the release versions you produce. If you move the plugins 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

12. Documenting your plugin

You can document your plugin's configuration options and behavior by placing a TEXT or RICH-TEXT (rtf) formatted file in the plugin directory. Give the file the same root name as your plugin and an .rtf extension (myplugin.exe uses myplugin.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 plugin 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').

13. Debugging Tips

1. Enable/Disable

Each time you recompile your plugin and wish to test it, you need to copy it to the plugin directory. If DigiView is still running and your old plugin 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 plugin, restart DigiView
- delete the signal using your plugin, copy the new plugin, recreate the signal
- disable the signal using your plugin, copy the new plugin, 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 plugin'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.

2. Avoid enabling more than one signal at a time that uses your plugin

When multiple signals use a single plugin, DigiView loads a single instance of the plugin into memory. DigiView will not unload the plugin 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 plugin. Also, each signal will attempt to use the plugin 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 plugin.

3. Task Manager

If Windows refuses to let you replace your plugin 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 plugin 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 plugin is disabled. If that is not the problem, then open the Windows task manager (usually available through cntrl-alt-delete) and review the Processes list. Find your plugin'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 plugin.

4. **Searches and Triggers**

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

5. **Debug option**

   DigiView includes an Environmental Setting to assist plugin debugging. Setting this option causes DigiView to modify its behavior as follows:

   1. **Pauses after loading a plugin**

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

   2. **Traps plugin communication timeouts**

      All interaction with a plugin is guarded with a 2 second watchdog timeout. Normally, if the plugin does not respond to a command or absorb enough data to allow new commands in that amount of time, an error is generated and the plugin is unloaded. When debugging support is enabled, a timeout dialog is presented rather than unloading the plugin. If you select 'CONTINUE', 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 'CONTINUE', DigiView will pick up where it left off rather than starting over.

6. **Streaming and Buffering**

   The communications between the DigiView application and your plugin 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 plugin 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 plugin will continue. If you were single stepping or you hit another breakpoint, DigiView will timeout and display the dialog again.

7. Common errors

1. Configuration string syntax

Fortunately, DigiView does extensive error checking of all of the configuration strings when the plugin is first loaded. Any errors are reported and the plugin is unloaded. Most error messages point to the specific field within the specific string with the error. Debugging this portion is usually pretty easy.

2. Field Chronology

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

3. 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 :)

4. 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 plugins will never receive raw data events and full plugins will never receive parser data events. Constantly checking would be a waste of time. Only hybrid plugins receive both types of events and need to differentiate between them.

If you take a mini or full plugin 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 plugin handles the event type code.

8. **Logging**

   DigiView does not currently provide any type of logging facility for plugin debug. However, your plugin 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 plugin, 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.

9. **Performance and stability**

   Keep in mind that your plugin becomes a part of the DigiView application. Its performance and stability affects the entire application. At any given time, there can be dozens of plugins operating. Also, each plugin is called to parse the raw data each time new data is captured. The plugin'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 plugin lockups or crashes. The plugin is loaded as a separate process. All communications with it use separate threads, timeouts (about 2 seconds) and large FIFOs. If the plugin 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 plugin lock ups, it could affect the stability of the DigiView application or the system.

14. Disclaimers and Restrictions

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

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.

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.

3. Use and Redistribution

You may use the files included in this Plugin Developers Kit to develop DigiView plugins 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 Plugin 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.
```

15. 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