Laboratory For Atmospheric and Space Physics
LASP Engineering Division
University of Colorado
Boulder, Colorado

Hydra User's Guide

Document No.

Approvers List

Prepared By	Elizabeth Cervelli, Flight Software Engineer

Rev	Change Description	By	Date
A	Initial release.	Beth Cervelli	

Table of Contents

Introduction

Purpose of this Document

The purpose of this document is to provide the HYDRA User with information on configuration and use of the application.

Document Scope

This document is limited to the use of the HYDRA application and its configuration files. No detail about the code implementation is provided.

Document Organization

Getting Started

Overview

HYDRA is a highly configurable platform for communicating with external devices via a variety of hardware interfaces. It has the following features:

- Bit level telemetry and command formats
- Real-time command interface
- Data display in numerical and graphical formats
- Data monitoring and notifications
- Scripting language with several script engines
- Built-in support for many types of devices (network, serial, files, etc)
- Data decoders for multiple types of packet formats
- Mix-and-Match decoders and hardware for easy customization
- Data archiving and playback capability
- Remote command capability

Below is a simplified view of HYDRA used as a telemetry decoder. Data comes in through hardware and is formatted into a frame by the Reader. The Reader then sends the frame to the Decoder, which uses a frame definition to decode the bytes into individual items and populate the Item Database with values. The display pages then read from the database and use the types, dictionaries and EU conversions to display the data on the screen. Monitors check items in the database for conditions and send notifications if needed.

This next diagram is of HYDRA used as a command terminal. Commands are entered by the user or sent from an executing script. A binary frame is constructed use the definition for the command and the inputs provided in the command string. The frame is given to the decoder, which sends it off to a hardware device.

Both types of data streams can be combined into one instance of HYDRA, and there can be multiple paths for telemetry and commands, with lots of different hardware interfaces. Decoders can be chained together to decommutate complicated packet formats. Decoders can also send frames to multiple hardware devices, or any combination of the above.

All of the boxes in the diagrams are defined and connected by the user in configuration files. Changing a hardware interface for a telemetry stream is as simple as hooking the reader into a different hardware device.

Main Window

At launch, the HYDRA application consists of a single main window, also called the event window. The window consists of an event area and a command prompt. Closing this window exits the entire program. Note that the window must be closed either by selecting the QUIT option via the File menu, or by Ctrl-W.

Event Area

The Event area of the window records HYDRA messages. Messages are generated when errors occur, commands are sent, or interesting things happen.

Messages are color coded to indicate status. Error messages are display in red, while input from the keyboard is displayed in blue. Positive message are shown in green. Other colors can come from item monitors (Section ###). Any messages shown in the window are also recorded in a log file.

Messages are preceded by an indication of their source, eg <KEY> for keyboard input and <MON> for monitor events. A list of all the source indicators is in ####.

The colors mentioned are the defaults and can be changed via config files (Section ###).

Command Prompt

The command prompt is located at the bottom of the main window. There are a number of commands that are processed internally by HYDRA. These commands can change state, start and stop processes, or print out HYDRA information. A complete list of HYDRA commands and their functions is in Section ###. The user can define other commands (Creating Commands) that are destined for hardware devices. Input that is not a command will simply be recorded in the event log.

Default Menus

The main window also provides a number of drop-down menus.

File

From the File menu, users can quit the program, save the current configuration (Section ###), or save the current window layout (Section ###).

Display Pages

Selecting a page from the Display Pages menu will open a window generated from a page definition file (Section ##). These windows update in real time to show the latest value of items in the HYDRA database. Only one window of each type can be open at once. Selecting an already opened window will simply bring it to the front.

View
The windows available under the View menu provide a look into the internal workings of HYDRA. They show the status of the items described in Section ###.

Summary
The windows available under Summary provide a look at the current status of HYDRA data flows and processes.

Event Messages
All declared event messages (Section ###) have entries in the Event Messages menu. Selecting a message opens a window that will display the content of the messages from that source. These windows only populate when they are open.

Command Menus
Any command that can be entered into the command prompt can be placed into a custom command menu (Section ###).

Project Directory
The directory where the HYDRA executable is installed must have a startup.xml file which contains a path to the desired project directory. If no path is provided, HYDRA will prompt the user for a directory. This directory contains all the configuration files HYDRA needs to setup and run for that project.

The project directory must have the following sub-directories: Pages (Section ###), Menus (Section ###), and Rundirs (Section ##). At startup, HYDRA will create a new folder under Rundirs with the current time, where it will store log files and archive files for that instance of the program. A new folder can be generated via HYDRA command.

Other than the required items, the user can organize the directory however they wish, as long as the correct relative paths are provided when needed.

Configuring HYDRA
Out of the box HYDRA does not do a whole lot of anything. It needs to be configured so it has telemetry and command formats, hardware interfaces, decoding info, etc. This is all done via the configuration files. All files are in XML format. The root node of the file needs to be <hydraDef>.

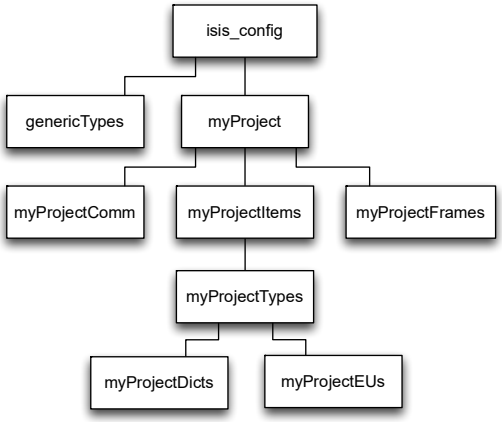
The organization of the files is up to the user. The only required file is hydra_config.xml, which HYDRA reads at initialization and must be in the lowest level of the project directory.

A file can specify other files to read.

```
<include>myConfigFile.xml</include>
```

Included files are completely processed before the remaining statements in the higher level file. This allows files to be nested and provide a tree hierarchy for organizing the information. The only rule about the configuration files is that if something is referenced in a file, it must have been declared previously, in the same file or a different one, otherwise HYDRA will not know how to handle the item.

Below is an example file hierarchy:



The following sections provide a detailed tutorial on how to create these configuration files.

Descriptions

Any configuration item described below can have a description associated with it, which provides more information about what the item represents than just its name.

```
<itemDef name="myItem" type="dn8">
      <description text="This is my very special item"/>
</itemDef>
```

This description will be displayed when viewing the configuration item via the in-program GUI.

Items and Types

Data is typically received from a hardware device in chunks of bits, called a packet or frame. The frame definition describes what each bit within the chunk represents, like a counter, temperature, voltage, or state. The process of extracting individual items from the frame is called decommutation. Frames that are received from hardware are usually called telemetry, while packets sent to a device to perform some action are called commands.

Declaring Items

HYDRA has a database of items, which are data structures that hold values. These items can be grouped into frame definitions, and as frames are received, the value of

each item is extracted and the database is updated with the new value. Items can be viewed and/or monitored for errors or other conditions.

To declare an item, its name and type are required.

```
<itemDef name="myItem" type="myType"/>
```

The name is used to uniquely identify the item in the database. The type describes how the data stored in the item should be interpreted, as well as the number of valid bits in the item.

If an item is not included in any frame definitions, it acts as a global variable and can be modified by the user in scripts or via the command line.

Declaring Types

Every item must have a type associated with it. The simplest type is the DN (data number). This type says that the bits in the item are to be interpreted as an unsigned integer with no conversion applied. This is also often called the "raw" value of the item.

```
<typeDN name="myDN" size="8"/>
```

This declaration creates an 8-bit DN type, which can then be assigned to any 8 bit items. The names of the types must be unique. There is no reason you cannot have multiple 8-bit DN types with different names, if that kind of organization is desired. The size of a type can range from 1 to 64 bits.

Endianness

For types larger than 8 bits, or types that can span byte boundaries, the endianness of the type becomes important. The endianness determines which of the bytes is treated as the most significant and least significant. This distinction becomes very important when working with different hardware architectures. Mac OS and MS Windows are little endian systems, while Unix and Linux are big endian. Multi-byte data sent between the two types will not be interpreted correctly unless the endianness is specified.

```
<typeDN name="myBigDN"    size="16" endian="BIG"/>
<typeDN name="myLittleDN" size="16" endian="LITTLE"/>
```

If no endian is provided, the default is used. At startup, the default is BIG, but it can be switched in the config files as needed.

```
<typeEndian default="LITTLE"/>
```

Other Native Types

Having everything declared as DN in the system is easy but limits the usefulness of the data. Telemetry such as counters are usually fine to display as DN, since they

are just integers that count up. But a temperature reported as DN will not be in degrees C or F, so it is not easy to tell what the data means. So HYDRA provides a number of other native types.

Signed DNs

The SN type is similar to the DN, but the bits are treated as signed instead of unsigned numbers. For example, an 8 bit unsigned DN has the range 0 to 255. But an 8 bit signed DN has the range -127 to 127.

```
<typeSN name="mySignedDN" size="8"/>
```

Floating Point

Floating point numbers are used to represent values that have a decimal point, like 2.77654. Floats are available in 32 and 64 bits and use the IEEE standard to interpret the bits.

```
<typeFloat name="myFloat"/>
<typeDouble name="myDouble"/>
```

Characters

Instead of treating an 8-bit value as a DN, it can instead turn it into a character, using the standard ASCII table.

```
<typeChar name="myChar"/>
```

Type Extensions

State Conversions

For some data, the value of the number is not important, but the state it represents is. For example, a hardware device could report a power state by sending a 0 if it is OFF and a 1 if it is ON. 0 and 1 do not mean anything by themselves, so HYDRA can apply a state conversion to turn the integers into strings.

To declare a state conversion, first you need to define a dictionary. These map integers to strings so HYDRA can find the string to display. A dictionary can be used by more than one type.

```
<dictionary name="myDictionary">
      <pair key="0" str="OFF"/>
      <pair key="1" str="ON"/>
</dictionary>
```

A type definition can then reference the dictionary, usually a <typeState>. Whenever the native or string value of any corresponding items is accessed, the dictionary value will be used. Otherwise, the data is interpreted as its base type.

```
<typeState name="myOnOffState" size="8" dict="myDictionary"/>
<typeState name="myOnOffState1Bit" size="1" dict="myDictionary"/>
```

These two types are nearly identical, but one will be used for 8 bit values and the other for single bit values. The dictionary does not need to change.

Image Maps

Instead of converting an integer to a string, like the previous section, an image map can convert the integer to an image. This conversion is only applied when an item is on a display page. The rest of the time the base value is used.

An image map is similar to a dictionary.

```
<imageMap name="myMap">
      <pair key="1" file="myImage1.bmp"/>
      <pair key="2" file="myImage2.bmp"/>
</imageMap>
```

The only supported image format at this time is bitmap (bmp). The map is then assigned to a type.

```
<typeDN name="myImageType" size="8" map="myMap"/>
```

Engineering Units

In many cases, a data value is the result of an ADC and represents a temperature or a voltage. An 8-bit value between 0 and 255 does not say much about what the temperature really is. A common conversion from the ADC value to temperature is to use a polynomial: $C_0x^0 + C_1x^1 + C_2x^2 .... + C_nx^n$ where the value is the X. The number of coefficients is up to the user.

```
<euPoly name="myEU">
      <coeff index="0" value="1"/>
      <coeff index="1" value="2.5"/>
      <coeff index="2" value=".006"/>
</euPoly>
```

Once the polynomial coefficients have been defined, a type can use it to actually convert values. Like dictionaries, multiple types can use the same polynomial.

```
<typeEU name="myEUType" size="8" eu="myEU"/>
```

Another possibility is to have a segmented EU. In this mode, the range of values is provided over which the polynomial is valid. For example, on negative values poly 1 is used, and for positives poly 2 is used.

```
<euSegmented name="mySegmentedType">
      <segment lower="-127" upper="0" eu="negPoly"/>
      <segment lower="0" upper="128" eu="posPoly"/>
</euSegmented>
```

There is no limit to the number of segments. For the ranges, the lower bound is inclusive (>=) and the upper is exclusive (<).

Arrays

Items can have multiple elements, which makes them arrays. Each element in the array is contiguous when extracting from a frame, and each element can be any number of bits wide. Array elements are accessed with the standard [] notation.

```
<itemDef name="myArray" type="dn8" num="5"/>
```

Initial Value

Items can be provided with an initial value. This is not used much by telemetry items since the value is overwritten with the first occurrence of its frame. But for standalone items and commands (Section ###) the value is set by the user. The default initial value is zero.

```
<itemDef name="myItem" type="dn8" value="4"/>
```

For array items, the initial value for each array element can be provided. If there are fewer values than elements, the remaining elements are set to the final value in the list.

```
<itemDef name="myArray" type="dn8" num="5" value="1 2 3"/>
```

The initial value of myArray will be [1 2 3 3 3].

If the item has a state conversion, the initial value can also be a string.

```
<itemDef name="myState" type="onOff8" value="ON"/>
```

History

Typically an item only stores the latest value. HYDRA provides an option to store last x values of the item in a history array.

```
<itemDef name="myItem" type="dn8" history="20"/>
```

With a history value of 20, the last 20 values for the item will be available to print or display on a page (Section ###). This is useful for items whose values update at a very high rate.

Derived Items

There is special kind of item whose value is derived from the value of other items by using arithmetic operations.

```
<itemDerived name="myDerived1" type="dn8" fact1="item1"
fact2="item2" opcode="+"/>
```

myDerived1 will be updated whenever item1 or item2 is updated, and its value will be the result of item1+item2. It does not make sense to add a derived item to a frame, since its value is not directly received.

The first factor of the equation must always be another item. But the second can also be a number, if for example we just want to add 1 to item1's value. The opcode can be one of the four standard arithmetic operators: +, -, *, and /. If using divide, ensure that the second factor will not be zero, otherwise HYDRA will complain about a divide by zero error.

Derived items do not necessarily need to be the same type as the factors. For example, the derived item could be a floating point number, or a state, while the factors are integers. Everything is converted into doubles before the arithmetic takes place. But they do need to have the same number of elements. It does not make sense to add an array of 5 integers to an array of 10 integers.

Accessing Item Values

There are several ways to view the current value of an item in the system.

Command Prompt

Any item can be viewed with the "print" command followed by the item name.

```
>>print myItem
>>print myArray[1]
>>print myArray
>>print_history myItem
```

The second example will print the second element of the array, while the third will print the entire array. The last will print the last x values of the item, if a history has been setup.

Display Page

Any item can be placed on a display page that will update regularly to show the latest value of the items. See Section ### for details on display pages.

View Menu

The Item menu under View allows the user to browse not only the value of items but their configuration info as well, and certain attributes can be modified while the program is running. Section ### provides more details on these menus.

Building Frames

A frame definition tells HYDRA where in each frame the value for an item resides, eg the fourth byte from the start. The definition also contains information that allows HYDRA to recognize that the chunk of bits it just received matches the given frame. Hardware devices usually either send one frame that always has the same format, or multiple types of frames that contain some kind of identifier.

Below is a straightforward frame containing two counters and a temperature.

```
<frameDef name="myFrame">
      <field>
            <itemDef name="counter1" type="dn8"/>
            <itemDef name="counter2" type="dn8"/>
```

```
            <itemDef name="temperature" type="dn16"/>
      </field>
</frameDef>
```

The frame's name is myFrame. The name is used by HYDRA to uniquely identify this definition. It contains the definitions of counter1, counter2, and temperature. The counters' both are 8 bits wide, as specified by the type, and the temperature is 16 bits. HYDRA assumes that these items are contiguous with no gaps; so all bits must be accounted for in the definition, even if they are not used by anything (pad, spare, or reserved names are usually used for these kinds of bits).

This is one way of declaring these three items. Another way is to declare them outside of the frame, and then reference them inside the definition.

```
<itemDef name="counter1" type="dn8"/>
<itemDef name="counter2" type="dn8"/>
<itemDef name="temperature" type="dn16"/>

<frameDef name="myFrame">
      <field>
            <item name="counter1"/>
            <item name="counter2"/>
            <item name="temperature"/>
      </field>
</frameDef>
```

In this case, the items counter1, counter2, and temperature must have been declared prior to HYDRA processing the frame, or it will complain that it does not know what the items are.

The first declaration method is most useful when an item only appears in one frame. It is clear which frame it comes from and if it changes it will only need to be updated in one place. The second definition is useful if the item appears in multiple frames.

```
<frameDef name="myFrame">
      <field>
            <item name="counter1"/>
            <item name="counter2"/>
            <item name="temperature"/>
      </field>
</frameDef>


<frameDef name="myOtherFrame">
      <field>
            <item name="temperature"/>
            <item name="counter1"/>
            <item name="counter2"/>
      </field>
</frameDef>
```

The new frame, myOtherFrame, has the same items, but they are in a different order. With this definition, the value of those items will update if either frame is received.

Sometimes it is nice to differentiate between values received from different sources. In that case, there are a couple of different options.

```
<frameDef name="myFrame">
      <field>
            <itemDef name="counter1_myFrame" type="dn8"/>
            <itemDef name="counter2_myFrame" type="dn8"/>
            <itemDef name="temperature_myFrame" type="dn16"/>
      </field>
</frameDef>

<frameDef name="myOtherFrame">
      <field>
            <itemDef name="temperature_myOtherFrame" type="dn8"/>
            <itemDef name="counter1_myOtherFrame" type="dn8"/>
            <itemDef name="counter2_myOtherFrame" type="dn16"/>
      </field>
</frameDef>
```

In this method, two sets of items are created, each with unique names that indicate which frame they are from.

```
<itemDef name="counter1" type="dn8"/>
<itemDef name="counter2" type="dn8"/>
<itemDef name="temperature" type="dn16"/>

<frameDef name="myFrame">
      <field>
            <itemCopy suffix="_myFrame" name="counter1"/>
            <itemCopy suffix="_myFrame" name="counter2"/>
            <itemCopy suffix="_myFrame" name="temperature"/>
      </field>
</frameDef>


<frameDef name="myOtherFrame">
      <field>
            <itemCopy suffix="_myOtherFrame" name="temperature"/>
            <itemCopy suffix="_myOtherFrame" name="counter1"/>
            <itemCopy suffix="_myOtherFrame" name="counter2"/>
      </field>
</frameDef>
```

This second method makes a copy of a master version of the three items and appends the provided suffix at the end, resulting in the same definitions as above. The advantage to doing it this way is that if counter1 was to change somehow, it only needs to be change in the master version, and it will be reflected in all its

copies. It is a little wasteful however, since the master copies are never updated, since they do not belong to any definition.

A shortcut exists to copy an entire frame, instead of each individual item. This is very useful if the items within the frame are changing often.

```
<frameDef name="myHeader">
      <field>
            <itemDef name="myId" type="dn8"/>
            <itemDef name="myLength" type="dn8"/>
            <itemDef name="myChecksum" type="dn8"/>
      </field>
</frameDef>

<frameDef name="myPacket">
      <field>
            <frameCopy name="myHeader" suffix="_pkt"/>
            <itemDef name="myPkt1"/>
      </field>
</frameDef>
```

This creates a copy of each item in the copied frame.

For itemCopy and frameCopy, prefix is also a valid tag for the new item names.

It is legal to use <itemDef> in one frame, and then <item> in another frame to reference it, but this creates readability issues as to which frame owns the item and it means the second definition always needs to be placed after the first. But HYDRA will not prevent this case.

Items that are declared but never used in a frame are still accessible to the user and scripts and as such can be used as global variables.

Absolute Positioning

The previous frame definitions all used relative positioning for their item definitions. In this mode, HYDRA assumes the first item is at position zero and uses the size of the item to determine the position of the next item. This is very helpful when a frame needs to be rearranged or items added because HYDRA will recalculate the positions automatically. But it is also possible to absolutely position items within a frame, or use a combination of the two.

The absolute position of an item is specified in the <field> tag.

```
<frameDef name="myFrame">
      <field startByte="1" startBit="4">
            <item name="firstItem"/>
            <item name="secondItem"/>
      </field>
</frameDef>
```

In this example, the starting position of the firstItem is given as byte 1, bit 4. The default values for both attributes is zero, if not provided. The position of the secondItem is still determined using relative positioning.

When using absolute positioning, the items do not need to be declared in any particular order and gaps in the frame where no values are defined is fine. There can be multiple <field> tags within a frame, even one per item if needed.

```
<frameDef name="myFrame">
      <field startByte="1" startBit="4">
            <item name="firstItem"/>
            <item name="secondItem"/>
      </field>
      <field startByte="10">
            <item name="middleItem"/>
      </field>
      <field startByte="20">
            <item name="lastItem"/>
      </field>
</frameDef>
```

Overlapping Items

Sometimes it is the case where a particular region within a frame has more than one way of interpreting it. An easy example is looking at a frame as just an array of bytes, and then also looking at each individual item. HYDRA provides a mechanism to allow both.

```
<frameDef name="myFrame>
      <field>
            <itemDef name="byteArray" type="dn8" num="10"
ignore="true"/>
            <itemDef name="item1" type="dn16"/>
            <itemDef name="item2" type="dn16"/>
            <itemDef name="item3" type="dn16"/>
            <itemDef name="item4" type="dn16"/>
            <itemDef name="item5" type="dn16"/>
      </field>
</frameDef>
```

In this example, the item byteArray is declared and its definition includes the ignore attribute. This attribute is only applicable inside of a frame definition, but can be used with any of the item definitions described in previous sections. All it does is tell HYDRA to ignore the size of the array when calculating the position of the next item, so the locations of both byteArray and item1 will be the same. The actual data from the frame will be copied twice, once into byteArray as 8 bit numbers, and then into each of the items as 16 bit numbers. An alternative to the ignore attribute is to use the absolute positioning method described above.

SubFrames

A sub frame is a frame that is nested inside of another frame.

```
<frameDef name="subFrame1">
      <field>
            <itemDef name="subFrameItem1" type="dn8"/>
            <itemDef name="subFrameItem2" type="dn8"/>
            <itemDef name="subFrameItem3" type="dn8"/>
      </field>
</frameDef>

<frameDef name="subFrame2">
      <field>
            <itemDef name="subFrameItem4" type="dn8"/>
            <itemDef name="subFrameItem5" type="dn8"/>
            <itemDef name="subFrameItem6" type="dn8"/>
      </field>
</frameDef>

<frameDef name="myFrame">
      <field>
            <item name="subFrame1"/>
            <item name="subFrame2"/>
      </field>
</frameDef>
```

This example shows two sub frames each containing three items that are then part of a larger frame. This could also be written this way:

```
<frameDef name="myFrame">
      <field>
            <itemDef name="subFrameItem1" type="dn8"/>
            <itemDef name="subFrameItem2" type="dn8"/>
            <itemDef name="subFrameItem3" type="dn8"/>
            <itemDef name="subFrameItem4" type="dn8"/>
            <itemDef name="subFrameItem5" type="dn8"/>
            <itemDef name="subFrameItem6" type="dn8"/>
      </field>
</frameDef>
```

Using subframes allows for more modularization when creating large telemetry packets and can make organizing all of the items easier. There is no limit to the level of nesting for these frames.

Frame IDs

Frames are uniquely identified by their name, but can also have an associated ID.

```
<frameDef name="myFrame02" id="2">
      <field>
            <itemDef name="item1" type="dn8"/>
      </field>
</frameDef>
```

This ID is used by the decoders to match frame definitions to received packets. If an ID is not unique, it must be associated with a group so the decoders do not get confused (Section ###).

The ID should match a field within the frame that is some fixed value. In the CCSDS header format, this would be the ApID. It is a field that identifies the remaining packet as having one format instead of another. To match the frame definition, a decoder is told that it can find the frame ID at location X within the bytes it is processing. It retrieves the value at that location and searches the database for the matching frame definition.

Stale Checking

A frame is considered stale when it has not been received for a while and therefore the value of the items in the frame may be out of date. To add stale checking to a frame, the maximum allowed interval between frames must be provided.

```
<frameDef name="myFrame02" id="2">
      <stale interval="10000" enabled="true"/>
      <field>
            <itemDef name="item1" type="dn8"/>
            <itemDef name="item2" type="dn8"/>
            <itemDef name="item3" type="dn8"/>
      </field>
</frameDef>
```

With an interval of 10000 ms, the frame will be declared stale if it is not received for 10 seconds. All of the items within the frame will be marked stale and, if displayed on a page, the display color will change. Setting enabled to false will turn off the checking for that frame.

Reading Telemetry

Once the frames and items are defined, the next step is to configure how the frames will get into HYDRA from outside the application.

HW Interfaces

HYDRA reads data through hardware interfaces. Each interface connects to a distinct type of device. The different devices available are TCP sockets, files, 1553 buses, serial ports, and OpalKelly FPGAs. Once defined, interfaces are interchangeable and the underlying details are not important.

To declare an interface, the type of device and the name of the interface must be provided, as well as any device specific configuration info.

Files

Files are a useful hardware interface, even though they are not really hardware devices. Output files are nice to use to record data from HYDRA so it can be checked for correctness, or read in as a data stream later. Input files are nice for simulating a data stream, since the data is predetermined and repeatable. And since HW interfaces are interchangeable, once a telemetry stream is working on a file, it will most likely work on the real stream, as long as the data formats are the same.

```
<hwInFile name="myInputFile" filename="myFile.in"
interval="1000"/>
```

This declaration creates an interface to an input file named myFile.in. This file is read only and cannot be written by this interface. The interval specifies the gap between reads. The gap makes the file look more like a serial port or a socket, with data coming in over time, instead of reading the entire file all at once.

```
<hwOutFile name="myOutputFile" filename="myFile.out"/>
```

This is a similar declaration for an output file. There is no need for an interval, because data will be written to the file as it is produced in real time.

An alternative declaration for an output file is to provide a prefix instead of a filename. In that mode, HYDRA creates the filename when the file is opened, based on the prefix and the current time. These files can be closed or rolled over by the user and each will have a unique name. Files with a prefix instead of a filename can also have a maxSize, which tells HYDRA to autonomously roll the file over when it hits that point.

```
<hwOutFile name="myRolloverFile" prefix="myData" maxSize="1000"/>
```

HYDRA will create a new file named myData_YYYY_DDD_HH_MM_SS every 1000 bytes of data.

-Handy Trick
When first trying to read a data stream, it can be very helpful to capture the stream in an output file and then use that file as the input, instead of the live stream. The data can be slowed down and/or visually inspected to find problems and get everything working.

Sockets

TCP/IP sockets come in two flavors, servers and clients. Server sockets are opened by HYDRA when it starts up and it waits for outside clients to connect. Clients must be opened by the user and they immediately try to connect to their server. HYDRA supports having both servers and clients running at the same time.

Sockets may be read-only, write-only, or duplex, which allows both operations.

```
<hwServer name="myReadServer" mode="read" port="2000"/>
<hwServer name="myWriteServer" mode="write" port="2001"/>
<hwServer name="myDuplexServer" mode="duplex" port="2001"/>
```

The above declarations create three server sockets. Only one client is allowed to connect to myReadServer and myDuplexServer, but there is no limit to the number of connected clients for the myWriteServer. Anything written to that interface will be broadcast to all connected clients.

```
<hwClient name="myReadClient" mode="read" port="3000"
addr="localhost"/>
<hwClient name="myWriteClient" mode="write" port="3001"
addr="localhost"/>
```

The next two declarations create client sockets that will attempt to connect over ports 3000 and 3001 to an application on a local machine.

Handy Trick
HYDRA applications on two different machines can talk to each other using server and client interfaces.

Serial Ports

Unlike files and sockets, all serial ports are duplex in nature. The same interface allows reading and writing. This makes sense because there is typically only one physical serial connection for a piece of hardware.

```
<hwSerial name="mySerial" port="COM1" baud="4800" parity="EVEN"
stopbits="1"/>
```

OpalKelly Devices

An OpalKelly FPGA is a commercially available device that provides a common interface to a customizable FPGA. The actual communication is via serial ports, but they are considered a separate HYDRA device. The DLL provided by the device must be available to HYDRA in order to talk to the device.

Once connected, HYDRA can send an optional configuration file to the device to set it up.

```
<hwOpalKelly name="myOpalKelly" config="myConfig.bit"/>
```

An OpalKelly device can be readonly, or duplex. For reading input data, a blockSize must be provided so HYDRA uses the appropriate read commands.

```
<hwOpalKelly name="myOpalKelly" readOnly="true" blockSize="256"/>
```

FIXME will have more config options later

1553

There are three types of entities that can exist on a 1553 bus: bus controller (BC), bus monitor (BM), and remote terminal (RT). As of this writing, HYDRA can only be setup as a bus controller.

This hardware implementation is specific to the GE family of 1553 controllers. The busapi DLL must be available and the appropriate drivers installed to use this interface. See ### for more details about the GE 1553 devices.

Bus Messages
The first step in setting up HYDRA 1553 communication is to define the message that will be sent across the bus. The message definition contains the remote terminal destination, sub address, size of message, and the type (Tx or Rx).

```
<hw1553Msg name="myCmdMsg" rt="1" subaddr="1" wcount="32"
type="TX"/>
<hw1553Msg name="myTlmMsg" rt="1" subaddr="2" wcount="32"
type="RX"/>
```

The first message is used for sending commands to RT #1 subaddress 1. Each message will have 32 words (64 bytes). The second message will request telemetry from the same remote terminal on subaddress 2.

HYDRA provides shortcut notation for messages that span several sub-addresses. The gap between the individual message transmissions must be provided in this definition.

```
<hw1553Msg name="myCmdMsg" rt="1" subaddr="1-4" wcount="32"
type="TX" gap="10"/>
```

Messages are their own hardware interfaces and are distinct from the bus controller interface. Unlike other interfaces, the bus controller itself does not send telemetry or commands; that is done by the messages instead. This means that when setting up a reader, the HW interface is the message, not the bus controller. Same for commands, the output device must be the message.

Bus Controller
The bus controller organizes the messages into a frame to create the message schedule. The duration for the frame provided is in microseconds, which is different from the HYDRA standard of specifying time in milliseconds.

```
<hw1553BC name="myBC">
      <minorFrame duration="1000000">
            <message name="myCmdMsg" start="1" repeat="1"
            gap="100" retry="true" bus="A"/>
            <message name="myTlmMsg" start="1" repeat="1"
            gap="100" retry="true" bus="A"/>/>
      </minorFrame>
</hw1553BC>
```

This BC has a 1 second minor frame, in which is sends each messages once during the frame. See the GE documentation for more details about the other configuration options.

In addition to the minor frame, the BC can have aperiodic messages, which are sent ad hoc when there is a gap in the schedule. This would be used for requests that do not need to happen on a fixed cadence, such as commands that are sent infrequently.

```
<hw1553BC name="myBC">
      <minorFrame duration="1000000">
            <message name="myTlmMsg"start="1" repeat="1"
            gap="100" retry="true" bus="A"/>/>
      </minorFrame>
      <aperiodic>
            <message name="myCmdMsg" start="1" repeat="1"
            gap="100" retry="true" bus="A"/>/>
      </aperiodic>
</hw1553BC>
```

Readers

Writing to a hardware device is pretty straightforward since HYDRA just writes data as it is generated using the interface provided by the specific hardware. Reading is more complicated. HYDRA needs to actively check the device for new things to read and somehow format that data into a frame to then be decommutated. Not all data streams are the same. Some send frames as discreet chunks, like socket connections. Otherwise send streams of bits with no clear demarcation from the hardware to separate the frames, like a serial port. The HYDRA object that makes sense of the stream is called a Reader.

A Reader's job is to gather frames from the hardware interface and send them to a decoder. If a Reader is not running, no data will come in.

```
<readerFixed name="myReader" size="1024" hw="myHW"
decoder="myDecoder"/>
```

The above declaration creates the simplest type of Reader, a Fixed Reader. It reads chunks of 1024 bytes from myHW and considers each chunk to be a frame. Once it has all the bytes, it sends them to myDecoder to be processed. It must be started by the user to start the stream (start_reader command). A Reader can be designated as "autoStart" instead, which means it is started as soon as it is created. To use this kind of Reader, the hardware must be in the correct state to be opened when HYDRA starts up.

```
<readerFixed name="myAutoReader" size="1024" hw="myHW"
decoder="myDecoder" autoStart="true"/>
```

This type of Reader works well for data streams where all frames are guaranteed to be the same size and communication is also guaranteed to be synchronous. A good example would be a TCP socket that only sends one type of frame. But it would not be a good fit for a serial port, where the Reader could presumably start its work somewhere in the middle of the stream. In that case the frame processed by the decoder would not start at the first byte and the decoder would extract the wrong values.

HYDRA provides a syncing mechanism to provide synchronous communication on data streams with a sync marker. The marker is some number of bytes at the beginning of each frame that is guaranteed to always be the same, and has a high

probability of being unique. IE whenever the sequence is found, it is almost always the marker and not somewhere in the middle of the frame.

```
<readerFixed name="mySyncedReader" size="1024"
decoder="myDecoder" hw="myHW">
      <sync size="4" pattern="0xFE 0xD4 0xAF 0xEE"/>
</readerFixed >
```

This new reader will still read frames of 1024 bytes, but it will make sure that the first four bytes match the provided pattern.

For devices with very high data rates, it can be an issue to read one or two bytes at a time looking for the sync pattern. If the reads are too slow, data could be lost. HYDRA provides a buffered read mode to help in these situations.

```
<readerFixed name="myBufferedReader" size="1024"
decoder="myDecoder" hw="myHW">
      <sync size="4" pattern="0xFE 0xD4 0xAF 0xEE"/>
      <buffer readSize="512" bufferSize="3072"/>
</readerFixed>
```

In buffered mode, the reader stores fixed size chunks of bytes in a buffer, and then the buffer is used to find the sync pattern and extract bytes. The read size does not need to be related to the frame size (512 vs 1024), but the buffer size should be at least 2.5 times the size of the frame to allow for proper syncing (1024 vs 3072). Of course buffered reads can be used without as sync pattern as well.

Other Reader Types

The previous section looked at a fixed reader. It defined a frame as having 1024 bytes. Fixed length frames are not always the case, so there are few other kinds of readers that construct frames using different methods.

Header Readers

This type of reader is used when all the frames in the data stream have the same header format, and the header contains a field that indicates the length of the total frame. In this case, HYDRA can read the bytes for the header, extract the length, and then use that to know how many additional bytes it should read to complete the frame.

```
<readerHeader name="myHeaderReader" headerSize="6" maxSize="1024"
decoder="myDecoder" hw="myHW" lengthField="2:0:dn8" offset="2"/>
```

For this kind of reader, the header size is provided as the minimum size of the frame. The max size indicates the largest possible frame. The offset says to add 2 to the length found to get the total length of the frame, including the header. The offset defaults to zero.

The location of the length within the header is indicated with a special notation HYDRA uses to specify the absolute position within a frame. The notation is startByte:startBit:type. In this example, the length is the third byte (numbering starts at zero) and is 8 bits wide.

There is an alternative to using this absolute position. Suppose there is a frame definition that describes the header:

```
<frameDef name="myHeader">
      <field>
            <itemDef name="FrameID" type="dn8"/>
            <itemDef name="FrameCount" type="dn8"/>
            <itemDef name="FrameLength" type="dn8"/>
      </field>
</frameDef>
```

The length field can then be written as:

```
<readerHeader name="myHeaderReader" headerSize="6" maxSize="1024"
decoder="myDecoder" hw="myHW"
lengthField="myHeader.FrameLength"/>
```

It still specifies the same location, but now if the header format changes and the length field moves or changes type, the reader is automatically updated. This only works if the myHeader frame is the outer frame, as HYDRA will get the position of FrameLength within just the myHeader frame. If there is another frame definition that contains myHeader, it also needs to be supplied:

```
<frameDef name="myFrame">
      <field>
            <itemDef name="FrameSync" type="dn32"/>
            <item name="myHeader"/>
      </field>
</frameDef>
```

and the reader would change to say:

```
<readerHeader name="myHeaderReader" headerSize="6" maxSize="1024"
decoder="myDecoder" hw="myHW"
lengthField="myFrame.myHeader.FrameLength"/>
```

This change puts the length in the 7th byte, due to the addition of the FrameSync field before the header.

This notation for absolute and relative positions is used throughout HYDRA config files.

TCP Reader

A TCP reader is designed to work with TCP connections. Frames sent over TCP are sent as complete transactions. Therefore, for this reader, only the max size of the

frame needs to be provided. When the socket is read, the number of bytes returned is assumed to be the length of the frame, and no inspection of a header is required. This is very useful for connections which pass strings back and forth.

```
<readerTCP name="myStringReader" maxSize="1024"
decoder="myStringDecoder" hw="myTCPPort"/>
```

Delim Reader

A delim reader looks for a delimeter character, which marks the begin/end of a frame. This character should not exist anywhere else the frame. FIXME expand

Mini Reader

A mini reader receives "mini" packets and constructs the larger frame using those. These packets are fixed length and contain sync markers to indicate if they are the start, middle, or end of the packet. FIXME expand

Decoders

As mentioned before, each reader sends the frames it gathers to a decoder for processing. More than one reader can send to the same decoder, or there can be lots of decoders for different frame formats. Readers and hardware have a one-to-one relationship however. Two different readers cannot read the same hardware at the same time.

A decoder's main job is to look at the frame it gets from the reader and match a frame definition to the bytes. It can then extract the items and update the database. There are two ways the decoder can find a matching frame. The first is the easiest:

```
<decoderDef name="myDecoder" updateDB="true">
      <frame name="myFrame"/>
</decoderDef>
```

This decoder assumes that all frames it receives match the definition myFrame. It doesn't have to make any decisions or inspect the bytes. This works great if there is only one type of frame.

The second method tells the decoder where in the frame it can find the ID of the frame, and then it goes to the database to find a frame with that ID.

```
<decoderDef name="myDecoder" updateDB="true">
      <frameID name="0:0:dn8"/>
</decoderDef>
```

Just as with the readers, the location of the ID can be specified using absolute (above) or relative (below) positioning.

```
<decoderDef name="myDecoder" updateDB="true">
      <frameID name="myFrame.FrameID"/>
</decoderDef>
```

The updateDB tag just tells the decoder that it is ok to update items using the values in the frame. If false, then the decoder will skip this step. A reason not to update the items might be that this decoder is receiving playback data and doing the update would interfere with the real time data coming down another stream. This type of decoder would probably just store the data instead.

```
<decoderDef name="myDecoder" updateDB="false" archive="true">
      <frameID name="0:0:dn8"/>
</decoderDef>
```

If archive is true and the decoder has matched a frame definition to its bytes, it will send the frame off to be archived to a file, but the archiving will only happen if there is one setup for the frame (Section ###).

Besides archiving and decommutating data, the decoder has one more job to do. It can also forward the frame on to hardware interfaces or other decoders for additional processing.

```
<decoderDef name="myDecoder">
      <outputDevice name="myOutHW"/>
</decoderDef>
```

The above decoder does not do any decommutating or archiving, its only job is to send the frames it receives to the myOutHW device.

```
<decoderDef name="myDecoder" updateDB="true">
      <frameID name="0:0:dn8"/>
      <outputDevice name="myOutHW"/>
      <outputDevice name="myOtherHW" frameID="2"/>
</decoderDef>
```

This next decoder sends all the frames it gets to myOutHW, but it also sends any frames that match id 2 to myOtherHW. There is no limit to the number of output devices.

```
<decoderDef name="myDecoder" updateDB="true">
      <frameID name="0:0:dn8"/>
      <outputDevice name="myHW1"/>
      <outputDevice name="myHW2" frameID="2"/>
      <outputDevice name="myHW3" frameID="3-5"/>
      <outputDevice name="myHW4" frameID="3"/>
      <outputDevice name="myHW4" frameID="4"/>
</decoderDef>
```

This last example declares that myHW1 gets all frames, myHW2 gets any that match ID 2, myHW3 gets the 3's, 4's, and 5's, and myHW4 gets the 3's and 4's.

In addition to HW interfaces, decoders can also send their frames to other decoders. There are lots of types of decoders that can perform further processing than just the type described so far.

Decoder Types

Header Decoder

A Header Decoder is almost identical to a regular decoder, but it removes a fixed number of bytes from the start of the frame before doing its processing.

```
<decoderHdr name="headerDecoder" length="4"/>
```

This example removes 4 bytes from the frame.

String Decoder

String decoders treat the entire frame as one long string, instead of as individual values. The contents of the string are written to a widow for viewing.

```
<decoderStr name="stringDecoder"/>
```

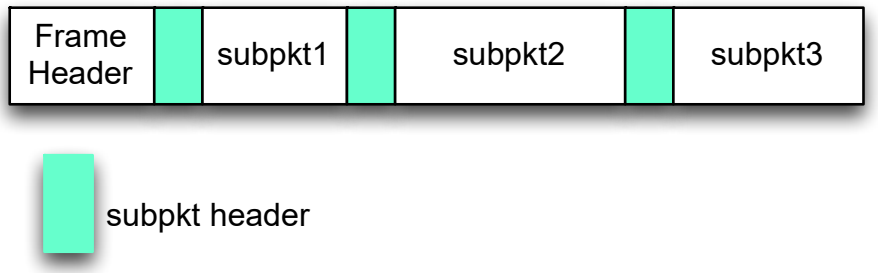
Command Decoder

A command decoder is like a string decoder; except it treats the strings it receives as an HYDRA command. This decoder allows for remote commanding of HYDRA over a TCP socket or similar.

```
<decoderCmd name="cmdDecoder"/>
```

SubPkt Decoder

Many packet formats involve subpackets in some way. These are smaller packets embedded inside a larger outer frame. The subpackets need to be extracted and processed the same as the larger frame is. Below is a figure showing what this data arrangement might look like:



Instead of sending each subpacket as its own frame, this application grouped the packets together into a larger frame with a different header. All the subpackets are different lengths and can come down in any order, so it would be impossible to write a frame definition to match the entire thing every time. The SubPkt decoder can retrieve each subpacket and process it as it would the larger frame.

```
<decoderSub name="subPktDecoder" startByte="12" lengthOffset="2"
subPktLength="0:1:dn16"/>
```

The startByte of the decoder tells it where the start of the first subPkt can be found. In this example, it would start right after the frame header. The subPktLength describes where in the subPkt the length field is, which is the same as how the header reader (Section ###) accomplishes its similar job. The offset is a value applied to the length to make it the size of the entire subpacket, including the subpacket header. This type of decoder only works if all the subpackets have the same header format.

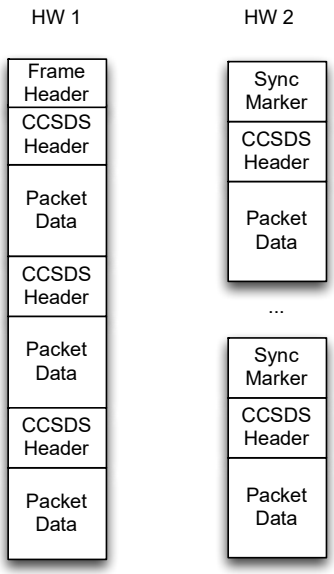
If all of the subpackets are guaranteed to be same length, an alternative declaration would look like this:

```
<decoderSub name="subPktDecoder" startByte="12"
subPktLength="256"/>
```

Combining Decoders

Here is an example of sending data between multiple decoders.

HYDRA is setup to read data from two different pieces of hardware, but both use the CCSDS format for their packets. The first application however sends its packets in transfer frames while the second prepends a sync marker to the front of each packet. The figure below shows the two different data streams:



Since both are using CCSDS packets, we would create a decoder to deal with those:

```
<decoderDef name="CCSDSDecoder" updateDB="true">
      <frameID name="0:5:dn11"/>
      <outputDevice name="ArchiveFile"/>
```

```
</decoderDef>
```

In addition to decommutating the frames, it also sends all the packets to an output file for safekeeping.

For the first data stream, we first want to decode the frame header and footer:

```
<decoderDef name="FrameDecoder" updateDB="true">
      <frame name="TransferFrame"/>
      <outputDevice name="SubPktDecoder"/>
</decoderDef>
```

The TransferFrame definition only has fields for the header, not the interior data. So those items would be updated and then the frame is shipped off to the next decoder, which is a subPkt decoder to extract the CCSDS packets from the frame:

```
<decoderSub name="SubPktDecoder" startByte="4" offset="0"
lengthField="2:0:dn8">
      <outputDevice name="CCSDSDecoder"/>
</decoderSub>
```

This decoder's only job is to extract the subpackets and send them for processing. It does not try to decommutate the frames at all.

An alternative to this would be to have the SubPktDecoder also process the CCSDS frames like so:

```
<decoderSub name="SubPktDecoder" updateDB="true" startByte="4"
offset="0" lengthField="2:0:dn8">
      <frameID name="0:5:dn11"/>
</decoderSub>
```

This is a totally legal and valid thing to do. But in this particular case we can use the same CCSDS decoder for two data streams and any changes to how these packets are handled would only need to be made in one place. Plus it helps with readability to have decoders with specific jobs.

The second data stream is easier; we just need to remove the sync marker, which we can do with a header decoder:

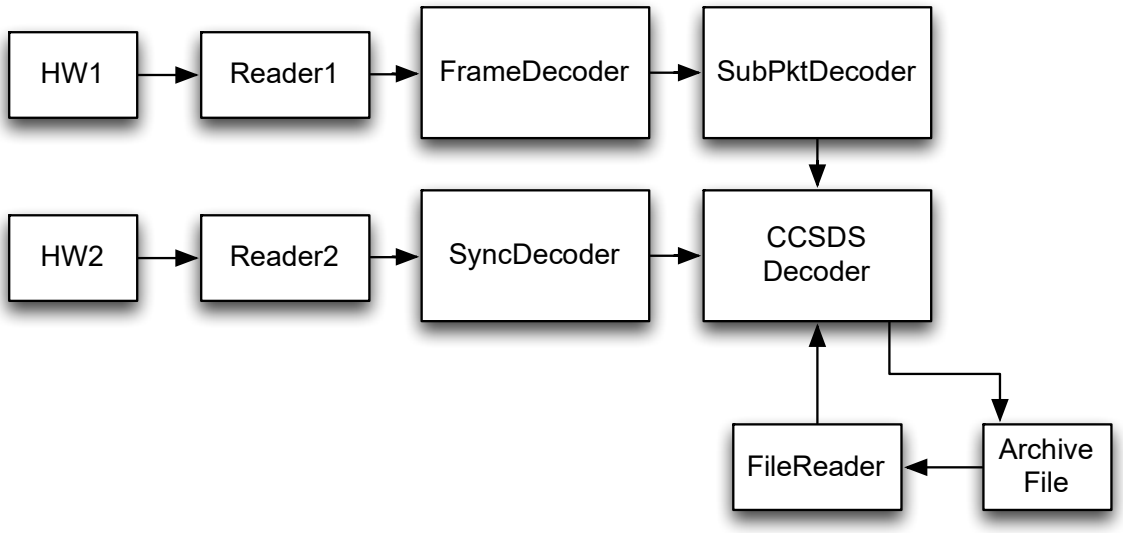
```
<decoderHdr name="SyncDecoder" length="4">
      <outputDevice name="CCSDSDecoder"/>
</decoderHdr>
```

Again, we could have made the header decoder process the CCSDS packets:

```
<decoderHdr name="SyncDecoder" updateDB="true" length="4">
      <frameID name="0:5:dn11"/>
</decoderHdr>
```

But we are trying to consolidate functionality by only using one decoder.

Another advantage to using the separate CCSDS decoder is that since it creates a file with just the CCSDS packets, another reader can send that data directly back to the CCSDSDecoder without making any other changes. The figure below shows the data flow for these three cases:



Hopefully you can see how combining decoders and hardware allows for very powerful and configurable processing of data streams.

Configuring the Project

Things to consider when applying the above to a specific project:

- What telemetry frames does HYDRA need to decode?
  - Which ones just need to be stored instead?
- Are there telemetry items that appear in multiple packets?
  - Should there be multiple copies of these items?
  - Or just one copy updated from multiple sources?
- Are there any telemetry items that are similar, but not the same, that making copies could be useful?
- What hardware devices will HYDRA connect to?
- What is the configuration info for those devices?
- What does the data stream look like?
  - Does the hardware send frames with headers?
  - Or a sync marker?
  - Does the reader need to be buffered for high data rates?
- How will the decoder know which frame it is processing?
- Should data be archived?
  - Forwarded to other hardware devices?

- Is it necessary to have multiple levels of decoding?

Creating Commands

Creating command definitions is similar to telemetry, but there are few more steps, so the user should be comfortable with item, type, and telemetry frame definitions before moving on to commands.

Commands are just frames, but they are created inside HYDRA and need to go somewhere. Commands also have specialized fields, like opcodes, and user generated values, like arguments.

All commands are sent to a decoder to be created. The most basic command decoder definition looks like this:

```
<decoderDef name="MyCommandDecoder">
      <outBuffer size="1024"/>
      <outputDevice name="MyCommandHW"/>
</decoderDef>
```

This is similar to the decoders from the previous chapter. The new thing here is the outBuffer. This is the place where the decoder will build the command frame so it must be as large as the largest possible command. A decoder without an outBuffer will complain if it is sent a command.

An outputDevice is necessary otherwise the command will be created but not go anywhere. Once the frame is built, it is processed just as described in Section ###, so command frames can be decommutated like telemetry frames if that is desired.

Basic Command

A simple command, like a noop, has no arguments and its values are fixed. Instead of using frameDef, we use the command tag. This puts the command into the official list and enables it to be invoked by name. But the rest of the definition is the same as a frameDef.

```
<command name="myNoop" decoder="MyCommandDecoder">
      <field>
            <itemDef name="noopOpcode" value="0" type="dn8"/>
      </field>
</command>
```

Instead of an ID, these frames have a decoder. The following definition

```
<command name="myNoop" decoder="MyCommandDecoder">
      <field>
            <opcode value="0" type="dn8"/>
      </field>
</command>
```

results in exactly the same bit pattern as the previous example. The only difference is that the opcode is called out as being special. There can only be one opcode field per command. This method results in a new item created with the command name + _opcode. The value of the opcode is also displayed on the command summary page. The second method is preferred because it adds more readability to the frame and prevents the user from needing to come up with a new unique name for every command. Of course, some commands don't have the concept of an opcode, and that is fine too.

Padding

The previous examples result in a single byte – 0 – being sent to the destination hardware. Sometimes, it is desired to pad every command to a certain number of bits or bytes.

```
<command name="myNoop" decoder="MyCommandDecoder">
      <field>
            <opcode value="0" type="dn8"/>
            <itemDef name="pad1" value="0" type="dn8"/>
            <itemDef name="pad2" value="0" type="dn8"/>
            <itemDef name="pad3" value="0" type="dn8"/>
      </field>
</command>
```

This example results in a padded command, but to specify pad items individually is tedious and can result in errors if the wrong number of items is provided. HYDRA provides a more succinct way of padding:

```
<command name="myNoop" decoder="MyCommandDecoder">
      <pad bytes="4" value="0xFF"/>
      <field>
            <opcode value="0" type="dn8"/>
      </field>
</command>
```

For this command, HYDRA calculates the number of padding bytes needed. This is very handy when commands are variable length. The value to pad with is optional, and defaults to 0 when not provided. This example results in the following sequence of bytes: 00 FF FF FF.

Protected Commands

A command defined as protected will pop up a confirmation box after it is invoked, but before it is created or sent. The user must hit OK to confirm that the command is desired. This is a useful for commands that can do potentially dangerous or undesirable things, like turn on components, or reboot the software.

```
<command name="myNoop" decoder="MyCommandDecoder">
      <pad bytes="4" value="0"/>
      <protected enabled="true" msg="Are you sure you want to
NOOP??"/>
```

```
        <field>
              <opcode value="0" type="dn8"/>
        </field>
</command>
```

If the enabled attribute is false, the command is treated as a regular command. The msg is optional.

Command Headers

For commands with headers, those fields can be placed into each command definition, just as any other field can be put into a frame definition. But that results in a lot of extra code, especially if there are lots of commands. Plus it makes it very hard to change the header, because hundreds of potential changes need to be made. HYDRA instead allows the header frame to be specified outside of the command definition, and then the commands can simply reference it.

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">
        <pad bytes="4"/>
        <field>
              <itemDef name="headerID" type="dn8" value="42"/>
              <itemDef name="headerCount" type="dn8" value="0"/>
              <itemDef name="headerLength" type="dn8" value="4"/>
              <itemDef name="headerChecksum" type="dn8" value="0"/>
              <placeholder/>
        </field>
</frameDef>
```

This is an example of a command header consisting of four bytes. Notice we used frameDef instead of command, since we would not invoke "MyCommandHeader" on its own. The padding has also moved from the command to the header. The new and interesting field is the placeholder. This indicates where the command should be placed into the frame. Any frame that is a header must include only one placeholder. If the size of the command is known, the number of bytes in the placeholder can be added as below:

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">
        <pad bytes="4"/>
        <field>
              <itemDef name="headerID" type="dn8" value="42"/>
              <itemDef name="headerCount" type="dn16" value="0"/>
              <itemDef name="headerLength" type="dn8" value="4"/>
              <placeholder size="256"/>
              <itemDef name="headerCRC" type="dn32" value="0"/>
        </field>
</frameDef>
```

HYDRA will then place the headerCRC after the 256th byte. If no size is provided, any fields that come after the placeholder are considered floaters, and their positions will be calculated when the entire frame is constructed.

Now that there is a header we can add it to our command:

```
<command name="myNoop" decoder="MyCommandDecoder">
      <header frame="MyCommandHeader"/>
      <protected enabled="true" msg="Are you sure you want to
NOOP??"/>
      <field>
            <opcode value="0" type="dn8"/>
      </field>
</command>
```

Everything is the same, except the padding has moved to the outer frame and we have a new <header> definition. Invoking this version of myNoop results in 8 bytes sent to the hardware, instead of 4.

An alternative to providing the header name in the command is to put it in the decoder instead. If all commands going to the decoder have the same header, this allows the header to be specified in just one place. This would not work if the decoder receives commands with different formats. Also, since the header for a decoder can be manipulated inside HYDRA at run time, the header for a command can easily be changed. See Section ### for more information on changing data streams dynamically.

```
<decoderDef name="MyCommandDecoder">
      <outBuffer size="1024"/>
      <headerFrame name="MyCommandHeader"/>
</decoderdef>
```

This is also useful for commands that might be sent over two different interfaces, each with a different header. By putting the header frame in the decoder for the interface, it lets the command definitions themselves remain the same. Below is an example of this configuration:

```
<decoderDef name="commandDecoder">
      <outBuffer size="1024"/>
      <headerFrame name="header1"/>
      <outputDevice name="device1" enabled="true"/>
      <outputDevice name="device2" enabled="false"/>
</decoderDef>

<command name="myNoop" decoder="commandDecoder"/>
```

When myNoop is invoked, the commandDecoder sends it to one of two devices, depending on which is enabled. To switch devices, the user just needs to change the header used by the decoder and then enable the correct device.

Command Length

So far the examples have just been for a command with one byte. Of course, in reality, commands come in all lengths. The header frame has a field called headerLength, which is supposed to specify the length of the command field after

the header. But since commands come in all lengths, hardcoding that value to 4 is not very useful. HYDRA can instead calculate the length dynamically and populate the field.

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">
      <pad bytes="4"/>
      <field>
            <itemDef name="headerID" type="dn8" value="42"/>
            <itemDef name="headerCount" type="dn8" value="0"/>
            <itemDef name="headerLength" type="dn8" value="0"/>
            <itemDef name="headerChecksum" type="dn8" value="0"/>
            <placeholder/>
      </field>

      <length item="headerLength" offset="-32"/>
</frameDef>
```

The length field tells HYDRA where to put the calculated length, and also if it should add/subtract an offset to the calculated value (in bits). So in this case, HYDRA will subtract 4 bytes from the total length, which accounts for the header bytes. The offset is optional and defaults to 0. This length is calculated after the command is created and all padding has been applied.

Any number of length fields can be supplied, and they can appear in any frame, not just header frames.

Command Checksum

The header example also has a checksum field, which is intended to be the checksum calculated over the entire frame. Since the length and command contents are dynamic, this calculation must also be done by HYDRA.

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">
      <pad bytes="4"/>
      <field>
            <itemDef name="headerID" type="dn8" value="42"/>
            <itemDef name="headerCount" type="dn8" value="0"/>
            <itemDef name="headerLength" type="dn8" value="0"/>
            <itemDef name="headerChecksum" type="dn8" value="0"/>
            <placeholder/>
      </field>

      <length item="headerLength" offset="-32"/>
      <checksum item="headerChecksum" start="headerID" stop="END"
seed="0xFF" method="xor"/>
</frameDef>
```

The checksum is similar to the length, in that it will populate the provided item with the result of its calculation. Instead of an offset, the user provides that start and end locations of the calculation, plus a seed value and the checksum method. When stop is END the checksum is all the way to the last byte. If instead, the stop is the name of

a field, the calculation will NOT include the stop field, ie it is not inclusive. There are several different checksum methods, see Section ###.

Also similarly to length, there can be multiple checksum fields and they can be in any frame definition. The checksum is calculated after the padding and all length calculations.

Counters

In this example, the headerCount field is meant to increment with each command so the hardware knows the commands are not out of order. To manually set this value every time would be tedious and error-prone. HYDRA provides a special type of item, called a counter that increments itself whenever it is put into a frame.

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">
      <pad bytes="4"/>
      <field>
            <itemDef name="headerID" type="dn8" value="42"/>
            <itemCounter name="headerCount" type="dn8" value="0"
            incr="1" min="0" max="128"/>
            <itemDef name="headerLength" type="dn8" value="0"/>
            <itemDef name="headerChecksum" type="dn8" value="0"/>
            <placeholder/>
      </field>

      <length item="headerLength" offset="-32"/>
      <checksum item="headerChecksum" start="headerID" stop="END"
      seed="0xFF" method="xor"/>
</frameDef>
```

Every time the header is used, the headerCount will increment by one and it will rollover when it hits 128 back to 0. The min and max values can be anything, and the increment can be 2, 4, or even negative.

Timestamps

Let's add a new field to our header:

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">

      <pad bytes="4"/>
      <field>
            <itemDef name="headerID" type="dn8" value="42"/>
            <itemCounter name="headerCount" type="dn8" value="0"
            incr="1" min="0" max="128"/>
            <itemDef name="headerLength" type="dn8" value="0"/>
            <itemDef name="headerChecksum" type="dn8" value="0"/>
            <itemDef name="headerTime" type="dn32" value="0"/>
            <placeholder/>
      </field>

      <length item="headerLength" offset="-32"/>
      <checksum item="headerChecksum" start="headerID" stop="END"
      seed="0xFF" method="xor"/>
```

```
</frameDef>
```

headerTime is supposed to be the time at which the command was created. This would be very hard to do manually, so HYDRA provides another special item type, called time.

```
<frameDef name="MyCommandHeader" decoder="MyCommandDecoder">
      <pad bytes="4"/>
      <field>
            <itemDef name="headerID" type="dn8" value="42"/>
            <itemCounter name="headerCount" type="dn8" value="0"
            incr="1" min="0" max="128"/>
            <itemDef name="headerLength" type="dn8" value="0"/>
            <itemDef name="headerChecksum" type="dn8" value="0"/>
            <itemTime name="headerTime" type="dn32" value="0"
            method="secSinceEpoch" epoch="Jan 1 2012 0:0:0"/>
            <placeholder/>
      </field>

      <length item="headerLength" offset="-32"/>
      <checksum item="headerChecksum" start="headerID" stop="END"
      seed="0xFF" method="xor"/>
</frameDef>
```

Now whenever this header is created, the number of seconds since the supplied epoch will be inserted into this field. There are different kinds of time methods available, see Section ###.

Command Arguments

HYDRA allows the user to specify the value of command fields when the command is invoked. These are the command's arguments.

```
<command name="myCmd1" decoder="MyCommandDecoder">
      <header frame="MyCommandHeader"/>
      <field>
            <opcode value="1" type="dn8"/>
            <argument id="arg1" type="dn8"/>
            <argument id="arg2" type="dn8"/>
      </field>
</command>
```

For the database, the argument id is added to the command name, so the user does not need to provide a unique name for every argument. This command would be invoked from the command prompt like so:

>>myCmd1 arg1 2 arg2 4

where 2 is the desired value for the first field and 4 for the second. If the type of the argument is not a dn, but a state, the name of the state conversion can be supplied instead and it will be converted to the appropriate numerical value. The same

applies to signed, float and char types. As of this writing HYDRA does not support converted EUs back to DNs.

A default value can be supplied for the argument, which is filled in whenever the argument is not provided at invocation. Arguments can also be range checked. If a value is supplied that is outside the allowed range, the command will not be sent.

```
<command name="myArg1" decoder="MyCommandDecoder">
      <header frame="MyCommandHeader"/>
      <field>
            <opcode value="1" type="dn8"/>
            <argument id="arg1" type="dn8" default="7"/>
            <argument id="arg2" type="dn8" min="1" max="5"
            minCheck="enabled" maxCheck="enabled"/>
      </field>
</command>
```

The range check can be for a lower bound, upper bound, or both.

Arrays

Arguments do not need to be single values; they can also be arrays.

```
<command name="myArgArray" decoder="MyCommandDecoder">
      <header frame="MyCommandHeader"/>
      <field>
            <opcode value="1" type="dn8"/>
            <argument id="arg1" type="dn8" num="5"/>
      </field>
</command>
```

The array values are populated by the user like so:

myArgArray arg1 [1|2|3|4|5]

where the | symbol separates each element. If all elements are not supplied, the remaining are populated with zeroes. FIXME or the last element?

Sending Queries

Overview

Queries ask for a specific status, i.e. the voltage of a specific channel, if a channel is ON/OFF, the system version. The response of the query is then returned to Hydra by the hardware, and stored in an item.

Constraints

In order for a query to work, the user must ensure that the hardware can handle queries. Currently, Hydra is configured to send queries via the SCPI (Standard Commands for Programmable Instruments) language. It is important that the user follows the syntax guidelines precisely, otherwise the command will not be parsed and sent correctly.

Creating a Query

There are two ways in which the user may send a query to the hardware: manually, or via the <query> item. For a full guide on how to format specific queries, refer to a SCPI datasheet.

Scripting a <query>

Because queries are meant to be sent on a regular interval, this should be the most common method used to send queries. The user creates a <query> item, specifying the name of the query, the SCPI string to be sent, and the item to store the query response, as well as the hardware, and decoder. For more detailed instructions on the xml format, please refer to <query>.

Note: The SCPI guidelines dictate that a *newline* (\n) must be added to the end of every query, in order for the query to be parsed correctly by the receiving hardware. Hydra internally adds the **\n** character to the end of each query, so it is not necessary to add it to the *string* tag within the query. It is however, very important that the user adheres to all spacing syntax.

Once the query item has been created, the user can send the *send_query* command, with the following format:

send_query <query name> [noPrint]

The user can either type this into the command line, or create a script that calls the send_query command in a while-loop. Either way, the command will update the *item* associated with the query response. The *noPrint* keyword is used to suppress the statements that are normally printed to the console. When manually sending a query, it is useful to see the print statements, but when queries are running continuously in the background, we do not necessarily need the print statements clogging up the console. Keep in mind, the *noPrint* keyword **only** works when used in a script.

Sending a query manually

This method is meant more for testing a query string, or if the user needs to run an infrequent, real-time check of the system.

In order to do this, the user utilizes the *send_str* command, with the following format:

send_str <hardware> <query string> newline

It is important to note that in the case, the *newline* string **must** be added at the end. Without this ending, the string will not be sent in the correct format to the hardware, and thus a response will not be sent back to Hydra. This method will print out the response in the reader window associated with the hardware. However, it will not update any item.

Configuring a Project

Before writing any command definitions, the user should identify where the commands are going to go. Is there a single hardware interface and all commands go to the same place? Or are there multiple hardware definitions with their own unique commands? Another possibility is that the same commands are sent to multiple interfaces at the same time. This information influences how the command decoders are setup.

As shown before in the tutorial, all commands must have a decoder designated as their destination. Otherwise, the commands cannot be created. The first possibility is easy, there is just one decoder that receives all commands and it sends them to one hardware interface. The third example is similar, one decoder, but it sends its commands to multiple interfaces. The second example requires multiple decoders. These can send to different interfaces, or the same ones. It just depends on the application.

The following figure shows how this might work:

Commands in group A are destined for devices 1 through N. Commands in group B are destined for device N, but also X and Y. An example of device N might be a file that is capturing all HYDRA output.

Decoders that are command destinations must have an output buffer specified. The size should be set to the length of the largest possible command. HYDRA will complain if a command is sent to a decoder without a buffer, because there would be nowhere for the frame to be constructed.

Once all the decoders and hardware are figured out, the commands can be created. If later it is necessary to add or remove a destination, only the decoder definition needs to change. Commands can be assigned to different groups simply by changing the destination decoder.

Advanced Topics

This section describes how to implement the more advanced features available in HYDRA.

Additional Decoders

The previous chapters discussed the following decoder types: basic, header, string, command, and subpacket. There are two additional types described below.

VCDU Decoder

A VCDU is a specific telemetry packet format used by many applications in the aerospace industry. The VCDU decoder is a variation on the subpacket decoder discussed before (Section ###).

A VCDU frame is fixed size and consists of a header, data section, and a CRC. The data section contains the subpackets. But since not all subpackets may be the same length, they may not all fit in the provided space. So the format allows a partial packet to be at the end of the section, with the remainder of the packet at the beginning of the next VCDU. Each VCDU has an incrementing sequence number, to ensure that the partial packets match up. For more information on the details of the VCDU format, see #####.

The VCDU decoder requires two additional pieces of information than the regular subpacket decoder.

```
<decoderVCDU name="myVCDUDecoder" subPktStart="12"
lengthOffset="0" subPktLength="vcdu.lengthField"
firstHdrField="vcdu.firstHeader"
sequenceField="vcdu.sequenceNum"/>
```

The sequenceField tells the decoder where it can find the sequence number, so it can ensure it is processing sequential frames. The firstHdrField tells it where it can find the first header pointer, which is how it knows where the first complete packet begins.

BCH Decoder

The BCH decoder is really more of an "encoder". Before sending the frame on to any output devices, it adds BCH codes to the data. For more information on the BCH algorithm see ####.

```
<decoderBCH name="myBCHDecoder" blockSize="8"/>
```

This is the only decoder that directly modifies the frame it is provided. This would not usually be used for telemetry processing, just commands.

Decoder Headers and Footers

There are two decoder options that apply to all decoder types (except string and command) that have not been mentioned yet. This is the ability to add a header and/or footer to the frame before it is sent to any outputs. They are really only used for commanding and they are distinctly different than the previously discussed command headers.

Of course, a header and/or footer can just be part of the frame definition for a command. But if there are lots of different frames sent to the decoder, but they all need the same header/footer, it can make sense to add them at the decoder level. It also means the header/footer will not be included in any length or checksum calculations performed on the frame. For the BCH decoder, the fields are added after the BCH codes are calculated, so they are also not part of that computation.

```
<decoderDef name="headerDecoder">
      <frameHeader type="dn8" num="4" value="0x01 0x02 0x03
0x04"/>
```

```
        <frameFooter type="dn8" num="2" value="0xFF 0xFE"/>
    </decoderDef>
```

The output buffer for the decoder must be large enough to accommodate the extra fields.

This is different than using the <headerFrame> attribute because the values are fixed, and no special frame processing can be done, such as applying padding.

Dynamic Data Streams

Sometimes it is necessary to change the source or destination of data in real-time. For telemetry, this is as easy as turning off one reader and turning on another. Commanding is a little more involved.

HYDRA allows the user to enable and disable output devices for a decoder on the fly and in the configuration files. For example, if commands can go to both HwA and HwB interfaces, but only one is desired at a time. The configuration file may look like this:

```
<decoderDef name="mySampleDecoder">
      <outputDevice name="HWA" enabled="true"/>
      <outputDevice name="HWB" enabled="false"/>
</decoderDef>
```

This configuration defaults to sending data to HwA. In order to send to HwB, it must be enabled by HYDRA command. HYDRA does not care if both are enabled/disabled. It is up to the user to disable one and enable another if exclusivity is important. The command to do this is:

```
>>set_dec_output mySampleDecoder HWA ALL disable
>>set_dec_output mySampleDecoder HWB ALL enable
```

The other thing that can be done on the fly is change which header is applied to the commands before they are sent. This is needed if different hardware devices have different communication protocols. This only works if the header is specified in the decoder, not the command itself (Section ###).

```
>>set_dec_header mySampleDecoder myNewHeader
```

Subfields

A subfield is a way to define a repeating item or frame within a frame definition. It is similar to subpackets, but the repeating items are all the same format. The number of repeats can either be fixed, or variable. If variable, there must be either a length or count indicator in the packet.

```
<frameDef name="myFrame">
      <field>
            <itemDef name="headerId" type="dn8"/>
```

```
            <itemDef name="headerCount" type="dn8"/>
            <itemDef name="headerData" type="dn8"
      subfield="count" control="headerCount" history="20"/>
      </field>
</frameDef>
```

In this example, the frame consists of an id field followed by a count. The rest of the frame depends on the value of count. If count is 5, then 5 copies of headerData will be processed and stored in the history of headerData.

```
<frameDef name="myFrame">
      <field>
            <itemDef name="headerId" type="dn8"/>
            <itemDef name="headerData" type="dn8"
      subfield="fixed" numFields="5" history="20"/>
      </field>
</frameDef>
```

This example shows a fixed subfield, in that there will always be 5 copies of headerData and there is no need for a control. This could also be accomplished with an array, but if the headerData is a frame, not an item, then an array would not work.

```
<frameDef name="myData" history="20">
      <field>
            <itemDef name="data1" type="dn8"/>
            <itemDef name="data2" type="dn8"/>
            <itemDef name="data3" type="dn8"/>
      </field>
</frameDef>


<frameDef name="myFrame">
      <field>
            <itemDef name="headerId" type="dn8"/>
            <item name="myData" subfield="fixed" numFields="5"/>
      </field>
</frameDef>
```

For this frame, the 3 fields of the myData frame are repeated five times in the larger frame. Declaring a history for a frame is just a shortcut for creating histories for all the items in the frame.

Besides count and fixed, the other control method is length.

```
<frameDef name="myFrame">
      <field>
            <itemDef name="headerId" type="dn8"/>
            <itemDef name="headerLength" type="dn8"
            <item name="myData" subfield="length"
            control="headerLength" offset="-2"/>
      </field>
```

```
</frameDef>
```

The number of copies of myData in this frame depend on the length field in the header, with an offset of -2. HYDRA calculates the length of myData, and divides the length with that number to get the number of copies.

So if the repeating data is all the same format, and the number of repeats is available or fixed, then this is an easy way to extract sub packets without needing to create a separate decoder.

Subfields can also be used in commands. The frame will use the available info on the number of repeats to construct a variable length packet.

Handy Trick
If the repeating item is a counter, it will increment every time it is placed into the frame, which makes it easy to create a command that contains a test pattern. Also, the controlling values do not have to be in the frame (ie global variables), they just have to be defined items so the value can be retrieved.

Frame Groups

Sometimes it happens that multiple frames coming into a system will have the same IDs. One way to mitigate this is to make the ID field larger to incorporate other fixed fields within the frame and therefore change the ID number. That is not always possible, so HYDRA allows you to assign frames to groups. A decoder can then be told to only look in a specific group for the frame definition. This solution only works on systems where the frames with duplicate IDs are processed by different decoders.

```
<frameDef name="myFrameGroup1" id="1" group="1"/>
<frameDef name="myFrameGroup2" id="1" group="2"/>

<decoderDef name="Group1Decoder" updateDB="enabled">
      <frameID name="0:0:dn8" group="1"/>
</decoderDef>

<decoderDef name="Group2Decoder" updateDB="enabled">
      <frameID name="2:0:dn16" group="2"/>
</decoderDef>
```

The group numbers are arbitrary and can be anything the user wants, except for zero. The zero group is the default group to which all frames are assigned.

Anytime there are duplicate IDs, the group attribute should be used to distinguish them. If the decoder cannot find the definition in the group list, it will look in the main list.

Senders

A Sender is an HYDRA process that sends a command at a fixed interval. The command is always the same, so to send different commands a script is more useful (Section ###). But senders are nice to send commands like noops every few seconds to verify a command connection is still enabled or a status message containing time and attitude info to a piece of hardware.

```
<senderDef name="mySender" frame="myNoop" interval="2000"/>
```

This declares a sender that will send the myNoop frame every 2 seconds. Senders are started and stopped by the user, and this type will run indefinitely until stopped.

To invoke a sender, all that is needed is the name. However, if the command requires arguments, they must be supplied at invocation.

The timing for senders is not guaranteed and they should not be used for time sensitive commanding.

Memory Loads

The sender mechanism is also used to facilitate sending memory loads to a piece of hardware. Memory load commands contain a buffer full of data that is destined for some location in the hardware memory map. The command may also contain an address for where the data should go, and/or the length of the buffer. The data for the buffer comes from a file and it can often take multiple commands to send the entire contents.

When started, a file sender reads the first set of bytes from the file, populates the command and sends it off. Then for the next command, it increments the address, grabs the next set of bytes, and repeats until all the desired bytes are transmitted. It accepts files in two formats, binary and Intel Hex-80.

```
<command name="myMemLoad" decoder="myDecoder">
      <field>
            <opcode value="1" type="dn8"/>
            <argument id="memAddr" type="dn8"/>
            <argument id="memLength" type="dn8"/>
            <argument id="memSection" type="dn8"/>
            <argument id="memBuffer" type="dn8" num="128"/>
      </field>
</command>

<senderFile name="myFileSender" frame="myMemLoad" interval="1000"
fileType="bin" length="memLength" addr="memAddr"
buffer="memBuffer"/>
```

The new parameters for this sender are the length, addr, and buffer. These map the sender fields to the fields in the destination frame. If the frame does not have a

length and/or address field, it is fine to leave them off. HYDRA will just not try and populate them.

To invoke the sender, a few more parameters than normal are required.

>>start_sender myFileSender filename myBinaryFile.bin total 1024 chunk 128 offset 0 memSection 2

The filename is just the name of the file to read from. The offset is the location within the file to start from. Total specifies the number of bytes to send and the sender will stop when it has read that number. The chunk size is the number of bytes to send per command. memSection is required because it is a parameter to myMemLoad that is not covered by the addr, length, or buffer fields in the sender itself.

Specifying all the parameters each time is tedious, so HYDRA provides default parameters when declaring the sender.

```
<senderFile name="myFileSender" frame="myMemLoad" interval="1000"
fileType="bin" length="memLength" addr="memAddr"
buffer="memBuffer">
      <defaults total="1024" chunk="128"
filename="myBinaryFile.bin" offset="0"/>
</senderFile>
```

Any of the defaults can be omitted, and they can be overridden by providing a value at the command line.

Sending a hex file is a little different than sending a binary file. Since the hex file contains address and length info inside it, HYDRA turns each row into a command, using the address and length for the row. The offset parameter in this case is not a byte location, but a line number.

```
<senderFile name="myHexSender" frame="myMemLoad" interval="1000"
fileType="hex" length="memLength" addr="memAddr"
buffer="memBuffer"/>
```

Archives

An archive is a file managed by HYDRA that is associated with a frame or an event message. Files are stored in the run directory and can rollover once they hit a certain max size. Archives must be specified inside the frame or message they are associated with.

A prefix for the filename is provided by the user. When a new file is created, HYDRA appends a timestamp in the format _YYYY_DOY_HH_MM_SS.

```
<frameDef name="myFrame">
      <archive prefix="myFrame" maxSize="1000" enabled="true"/>
```

```
      . . .
</frameDef>
```

In this example, a new file is created whenever the current one reaches 1000 bytes.

Timestamps

An archive can have an optional timestamp prepended to each entry in the file. The value of the time can be taken from another item value, or the Earth Received Time (ERT) can be used to record the system time, or both.

```
<archive prefix="timeArchive" enabled="true" ert="enabled"/>
```

Using ERT adds an extra 8 bytes at the front of each frame, while the size of the other timestamps is the same as their item type. HYDRA ERT format is seconds since 1970.

An alternative to the built-in HYDRA ERT is to declare a standalone item with a custom epoch and use that as the timestamp.

```
<itemTime name="myArchiveTime" method="secSinceEpoch"
type="dn32">
      <epoch year="2000" month="Jan" day="1" hour="0" min="0"
sec="0"/>
</itemTime>

<archive prefix="timeArchive" enabled="true"
timestamp="myArchiveTime"/>
```

This allows for timestamps in lots of different formats.

Archiving Vs Output File

This method of archiving is different from just writing frames to an output file, even though those files also support a max size and autonomous rollover. Output files are written by a device, while archive files are controlled by the frame. Therefore, if a user decides they always want to archive frame X, no matter its source, they can turn on archiving at the frame level, instead of supplying an output file for every decoder that might encounter frame X.

Event Messages

Imagine that a piece of hardware wants to send a message when something interesting happens. But it doesn't want to send plain text so it encodes the message using DNs. The resulting packet may look something like this:

```
<frameDef name="myEvent">
      <field>
            <itemDef name="eventID" type="dn8"/>
            <itemDef name="eventParam1" type="dn8"/>
            <itemDef name="eventParam2" type="dn8"/>
      </field>
```

```
</frameDef>
```

In this example, there is an ID number that maps to some message, and two parameters that are interpreted differently depending on the message. It would be nice to view this data as the actual message instead of the DNs. HYDRA provides a way to transform these data into readable text.

An event message uses a dictionary to turn the event ID into a string. If that is all that is needed, then it could just have the dictionary as part of its type. But what we really want to is add the parameters into the message, as well as potential other information like time received or other meta data. The event message describes how these items should be combined.

```
<msgDef name="myMsg" frame="myEvent" mainWindow="true">
      <item name="eventID" conversion="eventDictionary"
      param0="eventParam1" param1="eventParam2"/>
</msgDef>
```

This definition declares a message that is associated with the myEvent frame, so whenever it is received the message is created. It also says that the string should be printed to the main HYDRA window, in addition to the dedicated window provided for the messages (Section ###).

The item to print is the eventID, converted using the eventDictionary. That string is then fed the two parameters and it is all converted to text. Here is an example dictionary for this job:

```
<dictionary name="eventDictionary">
      <pair key="1" str="System state changed from %x to %x"/>
      <pair key="2" str="Address 0x%02x%02x accessed"/>
</dictionary>
```

The dictionary says that for eventID = 1, the two parameters represent system states and should be inserted where there are the %x codes. These are the same as printf style codes, so %x means print in hexadecimal. The parameters for eventID=2 are the MSB and LSB of an address instead. Up to four format codes are allowed per message.

You can define multiple strings for a message and HYDRA will concatenate them all together. They do not need to use a dictionary; they can be constants or just items with a little or no extra formatting.

```
<itemTime name="currentTime" type="dn32" method="secSinceEpoch"/>

<msgDef name="myMsg" frame="myEvent">
      <const str="Message arrived! "/>
      <item name="currentTime" format="%d: "/>
      <item name="eventID" conversion="eventDictionary"
      param0="eventParam1" param1="eventParam2"/>
```

```
</msgDef>
```

This definition results in messages that look like this:

Message arrived! 432465452: System state changed from 1 to 2

This formatting can be expanded to convert the parameter numbers into states themselves.

```
<dictionary name="eventDictionary">
      <pair key="1" str="System state changed from $state(%x) to
      $state(%x)"/>
      <pair key="2" str="Address 0x%02x%02x accessed"/>
</dictionary>
```

If "state" is another dictionary, the parameters in the first string will be converted to their strings. The $ character is a special character and should only be used to indicate a dictionary name.

Message arrived! 432465452: System state changed from IDLE to ACTIVE

Event messages can be used for other things than just doing this fancy conversion. An infrequently received frame could just write a message informing the world that it has arrived. It could be used to debug something, by printing info to the screen instead of just watching a display page.

Event messages can also be archived to their own file as text.

```
<msgDef name="myMsg" frame="myEvent">
      <archiveMsg enabled="true" prefix="myMessages"/>
      <const str="Message arrived! "/>
      <item name="currentTime" format="%d: "/>
      <item name="eventID" conversion="eventDictionary"
      param0="eventParam1" param1="eventParam2"/>
</msgDef>
```

Monitors and Notifications

Monitors are assigned to items to watch for a certain condition. When the value of the item meets the condition, the monitor triggers. This can result in messages written to the screen, display colors change, and/or emails or sounds to alert someone that something happened.

```
<monitorValue name="myMonitor" item="myItem" value="5" msg="red"
color="red" operator="=="/>
```

This declaration creates a value monitor assigned to myItem. The monitor name is not required, and if omitted, the name will be the item name + the type, so myItem_value. This means that there can only be one monitor of each type per item,

if explicit names are not provided. This particular monitor will trigger when the value of the item equals 5. It will print a message in red and change the display color of the item to red as well. These two colors do not need to be the same.

The message only happens once, when the monitor is triggered. As long as the value remains 5, the color will stay red. Once it changes value, the color returns to the default and the monitor waits to trigger again.

There are lots of different conditions that can be monitored for: equal, not equal, delta, change, no change, lesser than, greater than, in range, out of range, etc. A complete list of monitors is in Section FIXME.

Conditional Monitors

Sometimes we only care about monitoring an item if another item is in a certain state. For instance, you may only care about the temperature of a component if it happens to be powered on. For this case, HYDRA provides a conditional monitor.

```
<monitorState name="myPowerMon" item="myPowerState" state="ON"/>
<monitorValue name="myMon" item="myTemp" operator=">" value="10"
color="red" msg="red" condition="myPowerMon"/>
```

The myMon monitor will only trigger if myTemp is greater than 10 AND myPowerState equals ON. If the power state changes, myTemp is no longer triggered.

Since we don't really care about getting a message every time the power state changes, myPowerMon can be set to silent. It will still trigger, but it will not annoy the user with messages.

```
<monitorState name="myPowerMon" item="myPowerState" state="ON"
silent="true"/>
```

Notifications

By default, a monitor only writes messages to the screen and changes display colors when triggered. If that is not enough, a Notification can be associated with the monitor and it will take additional action.

```
<notifyEmail name="myEmailNotify" addr="myEmail@email.com"/>

<monitorValue name="myMonitor" item="myItem" operator="=="
value="5" msg="red" color="red">
      <notify name="myEmailNotify"/>
</monitorValue>
```

Now when myMonitor triggers, it will send an email to the provided address with the trigger info. A monitor can have an unlimited number of notifications, if lots of people want to be emailed.

Another type of notification is to play a sound file.

```
<notifySound name="mySoundNotify" filename="mySound.wav"/>
```

Each monitor should really only have one sound type notification at any time, otherwise the sounds try to play on top of each other and things can break.

Notifications can be associated with frame stale checking as well as monitors.

```
<frameDef name="myFrame">
      <stale interval="1000" enabled="true"
notify="staleNotification"/>
</frameDef>
```

In this case, only one notification is allowed per stale check.

Red/Yellow Limits

Red/Yellow limits are a special type of monitor. The range of values for the item is divided into 5 possible zones – red low, yellow low, green, yellow high, and red high. The colors associated with the message and telemetry display match the color of the zone for the current value. Messages are provided at every transition between zones, but notifications are only sent for transitions to red and yellow zones, not green.

```
<monitorRedYellow item="myItem" rl="1" yl="5" yh="10" rh="15"
enabled="true"/>
```

If one or more of the values is omitted, that zone is not used. For this example, the item is green when it is between 5 and 10, red when less than 1 or greater than 15, and yellow any other time. All of the comparisons used are exclusive (>, <).

A side-effect of this type of monitor is that it is always in the triggered state, so it is cannot be a condition for another monitor. It also cannot be set to silent.

Stale Monitors

A stale monitor is another special kind of monitor. Once created, the stale monitor checks that value of an item every second to see if it has changed. If not, the monitor increments a persistence counter, and if the counter goes above the allowed value, the item is declared as stale.

```
<monitorStale item="myItem" interval="10000" enabled="true"/>
```

The interval for the monitor is the max number of milliseconds between value changes.

This behavior is similar to the no-change monitor, but it does its checking based on time, whereas the no-change monitor checks the value each time the item is received. A nice use for this type of monitor is making sure a data stream is on by

looking at the number of bytes read by a reader. It can also check for stale frames by checking the frame count for said frame.

Gumballs

A gumball is a collection of items that are being monitored. There are three possible states for the gumball – green, yellow, and red. A message is printed whenever the gumball transitions, and they can be displayed on Display Pages (Section ##).

The gumball is green whenever all of the items in the gumball are in a "good" state. This is the default state.

The gumball is red whenever any of the items in the gumball are in a "bad" state. This condition is triggered when a monitor for the item is triggered AND the desirability of the monitor is designated as bad. The item will not return to a good state until all of its monitors are also in a good state.

The gumball is yellow whenever any of the items are in a "caution" state AND there are no items in a "bad" state. Like the "bad" state, this occurs when a monitor for the item is triggered AND the desirability of the monitor is caution.

Red/Yellow monitors are the only monitor type with implicit good/caution/bad states: yellow zone is caution, red zone is bad, and the green zone is good. The desirability of other monitors must be explicitly defined.

```
<monitorValue item="myItem" enabled="true" operator="=="
value="5" desire="bad"/>
```

The following monitor types do not have a desired state and cannot be used to trigger a gumball: change, no change, delta, and update.

Gumballs are defined in a configuration file and are referred to by name in the display page definition file (Section ###). An item can belong to more than one gumball.

```
<gumball name="myGumball">
      <item name="myItem1"/>
      <item name="myItem2"/>
      <item name="myItem3"/>
</gumball>
```

Saving Database as XTCE

Hydra is able to write XML files in the LASP XTCE format for command and telemetry definitions. There are two input files required for this function, a template file and an input file.

The template file is an empty file generated from XTCE-Editor. It contains the CCSDS packet headers for telemetry packets and commands. The input file is a user created XML file that contains a list of all the telemetry and commands that should be translated. It has the following format:

```
<hydraXTCE>
	<tlm>
		<packet>packet_name</packet>
		<packet>parent.child_name</packet>
	</tlm>
	<cmd>
		<name>my_cmd</name>
	</cmd>
</hydraXTCE>
```

When listing a telemetry packet, either the packet on its own can be provided, or a packet and parent parent is provided. The translator assumes that the packets contain the CCSDS header and will encapsulate the packets in the definition as defined by the template file. Therefore, the packet being processed should not have its CCSDS headers defined in that packet. The parent is meant to be the larger packet that contains the child and CCSDS headers. Its ID will be used as the ApID of the packet for the translation. An example of this is below:

```
<frameDef name="my_frame">
	<field>
		<itemDef name="item1" type="dn8"/>
		<itemDef name="item2" type="dn8"/>
	</field>
</frameDef>

<frameDef name="my_parent" id="0x1ff">
	<field>
		<frameCopy name="ccsdsPriHeader" suffix="_1"/>
		<item name="my_frame"/>
	</field>
</frameDef>
```

For this example, if the translation is called for just my_frame, the ApID used will be -1 as that frame does not have an ID. If called for my_parent.my_frame, it will use the ApID of the parent. Translating my_parent by itself will result in two CCSDS headers defined for the packet, which would be incorrect.

When calling the translation, in addition to the input files listed above, an output file should be provided that will hold the translated definitions.

>>save_xtce templateFile inputFile outputFile

Internal HYDRA Items

The user defines most of the items in the HYDRA database, but HYDRA also creates its own items to act as housekeeping information for readers, decoders, frames, etc.

These can be accessed the same as any other item and placed on display pages, assigned monitors, and even added to frame definitions. The name of the item is created from the name of the object it is monitoring and its function. Below are the current internal items.

General
HYDRA_cmdCnt – number of internal HYDRA commands processed since initialization

Readers
ReaderName_bytesRead – number of bytes received by the reader since initialization

Frames
FrameName_recvCount – number of frames received
FrameName_sentCount – number of frames sent as a command

Senders
SenderName_status – 0 for IDLE, 1 for running state

The size of all of the items is 32 bits.

Handy Trick
If another application wants the status of HYDRA, the relevant internal items can be added to a frame definition and sent at a regular cadence by a sender to the outside application.

Command Aliases

HYDRA allows the user to create aliases for commands that have preset values for certain parameters. Doing this gives the illusion of more commands for a system, without needing to redefine the opcode and other fields for each one. It can be used as a shortcut for commonly used command/parameter combinations.

```
<alias name="turn_off_channel_2" command="turn_off_channel">
      <argument id="channel" value="2"/>
</alias>
```

The above alias creates a new command "turn_off_channel_2" that automatically fills in the channel parameter of the more generic "turn_off_channel" command. If the command has more parameters, they will still need to be supplied when the alias is invoked.

Aliases can be created for internal commands as well. These do not take name/value pairs for parameters, so the argument id should be the position number of the argument value for the particular command. Because the parameters are position dependent, all parameters must be present in the alias.

```
<alias name="rollover_file" command="cmd_hw">
      <argument id="0" value="myFile"/>
      <argument id="1" value="rollover"/>
</alias>
```

Color Schemes

HYDRA has a default set of colors that it uses to display messages and items on display pages. These can be modified by redefining the color definitions in the config file. The following colors are available to change:

Color Name	Default	Use
ERROR	red	Error messages, red limit violations
GOOD	green	Status messages, no limit violations
CAUTION	yellow	Caution limit violations
STALE	purple	Packet stale monitor has expired
KEYBOARD	blue	Keyboard input

To change a default color, a new code for the color must be supplied. This can either be a color name, or an rgb code.

```
<color name="GOOD" code="BLUE"/>
<color name="BAD" code="rgb(0,255,0)"/>
```

The default settings for display pages and the event windows can also be changed in the same way.

Color Name	Default	Use
EVENT_WIN_BACK	white	Background for event message window
EVENT_WIN_TEXT	black	Default text color for event messages
DISPLAY_PAGE_BACK	gray	Background for display pages
DISPLAY_PAGE_TEXT	black	Text color for display pages

Creating Displays

Any declared item can be shown on a display page, including HYDRA internal items and global variables. These pages update on a regular cadence, default is 1Hz, to show the current value of the item. State conversions and EU conversions are automatically applied to the value. The Items can also change color if a monitor is triggered for the item.

The definition of a page is in an XML file that must reside in the Pages directory in the HYDRA run directory. Only one page can be defined in each file. HYDRA reads the directory at startup and populates the Display Pages menu with the contents of the directory. Individual pages are read when opened, so an existing page can be changed without a restart.

New page definitions can be created within HYDRA using the “create_page” command. This command populates the file with the contents of a defined frame. The new page is saved to the Pages directory with the same name as the frame. It is also added to the Display Pages menu for viewing. If the frame does not exist, it creates an empty file.

Page Definition Basics

All definition files have <pageDef> as their root node, followed by the <page> node.

```
<pageDef>
	<page title=”myPage”/>
</pageDef>
```

The title of the page is displayed on the top bar of the window and should match the filename, without the .xml.

The display window is divided into columns. HYDRA determines the spacing of each column as it builds the page, so it will dynamically adjust the spacing as needed. Each column is placed on the page in the order they appear within the file.

```
<pageDef>
	<page title=”myPage”>
		<column/>
	</page>
</pageDef>
```

The page can have as many columns desired. The window will have scrollbars that let the user access any columns that aren’t displayed due to the window size.

Everything else on a page is placed within a column. Nothing in the file has an absolute position defined. Everything is defined relative to everything else. HYDRA takes care of determining how wide and long each component needs to be in order to display the entire string.

Sections

A section is a set of rows that are grouped together. Each row in the section can have a label, item name, or both. The item name must match the name in the database, but the label can be anything.

```
<pageDef>
	<page title=”myPage”>
		<column>
			<section name=”section1”>
				<row label=”firstItem” item=”item1”/>
				<row label=”secondItem” item=”item2”/>
				<row label=”thirdItem” item=”item3”/>
				<row label=”fourthItem” item=”item4”/>
			</section>
		</column>
	</page>
```

```
</pageDef>
```

This definition results in a page that looks like this, assuming the values of items 1 – 4 are 1, 2, 3, and 4.

	section1
firstItem	1
secondItem	2
thirdItem	3
fourthItem	4

If one of the items has a state or EU conversion, the converted value is shown instead.

Omitting either the label or the item in the row just leaves a gap where that element would have been.

Units can be added to the item by declaring them in the row definition:

```
<section name="section1">
      <row label="firstItem" item="item1" units="C"/>
      <row label="secondItem" item="item2"/>
      <row label="thirdItem" item="item3" units="ms"/>
      <row label="fourthItem" item="item4"/>
</section>
```

	section1
firstItem	1 C
secondItem	2
thirdItem	3 ms
fourthItem	4

The default behavior of the page is to display the item as its converted value. The other possible formats are hex, binary, raw, and custom.

```
<section name="section1">
      <row label="firstItem" item="item1" format="HEX"/>
      <row label="secondItem" item="item2" format="BINARY"/>
      <row label="thirdItem" item="item3" format="RAW"/>
</section>
```

A shortcut for displaying items is to provide only the item name. The name will then be used as the label.

```
<section name="section1">
      <item name="item1"/>
      <item name="item2"/>
      <item name="item3"/>
      <item name="item4"/>
```

```
</section>
```

This results in the following display:

section1
item1 1
item2 2
item3 3
item4 4

An optional display name can be provided with an item definition. If provided, this label will be used in this case instead of the item name.

```
<itemDef name="myItem" type="dn8" displayName="My Item"/>
```

section1
My Item 1
item2 2
item3 3
item4 4

Another shortcut is to just provide the frame name. This will automatically create a section with all the items in the frame. Very useful if a frame changes frequently, as the display page will automatically update with the frame definition.

```
<column>
      <frame name="myFrame"/>
</column>
```

myFrame
My Item 1
item2 2
item3 3
item4 4

Monitors

The status of a monitor can be placed in a row instead of an item. The display consists of a colored box that corresponds to the color of the specific monitor.

```
<row label="myMonitor" monitor="myMonitor"/>
```

Nested Sections

It is legal to have a section inside of another section. This results in the nested section displayed above the outer section. They do not appear "nested" on the page. This is really only used by the automatic page creation command (Section ##).

When making a definition from a defined frame, subframes are added as nested sections within the larger frame.

Item History

Instead of using a row to show the latest value of an item, the entire history of the item can be displayed instead. This does not do anything for items that do not have a history configured (Section ###).

```
<section name="section1">
      <row label="item1" item="item1"/>
      <history item="item2"/>
</section>
```

The last x values of the item will be displayed as a column of numbers with the most recent value on top.

Arrays

If an item is an array, individual elements can be placed in rows using standard array notation. Another option is to display the entire array at once.

```
<section name="array">
      <array item="myArray" rows="8" cols="10" format="DEC"/>
</section>
```

The number of columns defaults to 16 and the number of rows to zero. If no rows are provided, HYDRA will calculate the number of rows need to display the entire array given the number of columns. If the total number of elements does not encompass the entire array, the display will only show the first elements that do fit. The two valid formats for displaying the array are DEC (DN) or HEX.

Spacers

A blank line can be inserted into a section with a <spacer/> tag.

Items

Items can be added directly to columns instead of using a section. As with a row, the item can specify its units and format info.

```
<column>
      <item name="myItem" units="ms"/>
</column>
```

Text

Text strings can be added directly to a column.

```
<column>
      <text str="this is my string"/>
</column>
```

Tables

Instead of displaying items in row/label pairs, a table construct is available. The number of rows and columns must be supplied. The row and column names are supplied, and then the items for each row/column.

```
<column>
      <table>
            <tableTitle name="myTable"/>
            <tableHeading>
                  <label name="A"/>
                  <label name="B"/>
            </tableHeading>
            <tableRow>
                  <label name="Level"/>
                  <item name="itemALevel"/>
                  <item name="itemBLevel"/>
            </tableRow>
            <tableRow>
                  <label name="Value"/>
                  <item name="itemAValue"/>
                  <item name="itemBValue"/>
            </tableRow>

      </table>
</column>
```

Monitors

Monitors can be added directly to a column.

```
<column>
      <monitor name="myMonitor"/>
</column>
```

Buttons

A push-button can be placed anywhere within a column, but not within a section. The button has a label and command string associated with it. When pushed, the command string is executed just as it would be if it originated from the command line.

```
<column>
      <button name="myButton" cmd="myNoop"/>
      <section name="mySection>
            <row label="item1" item="item1"/>
      </section>
</column>
```

Charts

A chart is a way to display items graphically, instead of numerically. The charts provided by HYDRA are plots of time vs item value. They update at a regular cadence, even if the value of the item is not changing.

```
<column>
```

```
        <chart title="myChart" update="1000" interval="120000">
              <line item="myItem" color="blue"/>
              <line item="myItem2" color="red"/>
        </chart>
  </column>
```

This chart updates with a new value every second (1000ms), and stores values for 2 minutes (120 seconds). It has two items plotted on it, one in blue and the other in red. Both items are plotted on the same axis, so if more than one item is included, they should have roughly the same scale.

It is also possible to set the min and max values for the y axis. Otherwise, the plot is auto-scaled to include the entire line.

Images

An item that has an image map for its type can be displayed on a page as its image conversion instead of its raw value. This is not automatic: HYDRA must be told that the item has an image type. Also, these items cannot be displayed in a section, they must be in a column.

```
<column>
      <image name="myImageItem"/>
</column>
```

HYDRA will complain if the item is not actually an image type.

Gumballs

Gumballs can be placed anywhere within a column. The display item consists of the gumball name and a colored circle representing the state. There are three possible states for the gumball – green, yellow, and red (Section ###). When yellow or red, the number of items in that state is displayed in the center of the circle.

```
<column>
      <gumball name="myGumball"/>
</column>
```

If a gumball is clicked, a new page is displayed that contains the value of all of the items from the gumball. This page is auto generated by HYDRA and is not defined anywhere in a file. Clicking more than once will result in multiple copies of the same page.

Creating Pages Automatically

HYDRA can auto-generate a page definition file for a specific frame. All items in the frame will be placed into rows in one section inside one column, with the label matching the item name or display name. The resulting page is not very readable, as it will be just one long list of items, but the file can then be easily edited by hand to add column breaks, new sections, change label names, add buttons, etc.

Any nested frame definitions inside the frame will be put into nested sections in the definition file.

>>create_page frameName

The result will be a file in the Pages directory with the name frameName.xml. If there was already a file with that name, it will be overwritten. The Display Pages menu will then be updated with the new page name, so it can be opened and viewed immediately. If the frameName does not exist, a blank file with only the <pageDef> and <page> nodes will be created.

Changing Page Defaults

All pages have a default update rate, size, and screen position. But they can be changed on a page-by-page basis.

```
<pageDef>
      <updateRate ms="2000"/>
      <size width="200" height="400"/>
      <position x="0" y="20"/>
</pageDef>
```

The background color, font, and font color can also be changed on a page-by-page basis.

```
<pageDef>
      <background color="BLUE"/>
      <text color="WHITE"/>
      <font name="Arial" size="12"/>
</pageDef>
```

Using Display Pages

When selected from the Display Pages menu, the definition file is read and the page is populated with the defined columns, sections, and rows. If the window is already open, it is brought to the front instead of a new window being created. The page will update at its update rate indefinitely until closed. To change a page, just close the window, update the definition file, and reopen the window.

Display Format

Right-clicking an item on a page brings up a menu that allows the user to change the display format of the item to RAW, HEX, BINARY, and NATIVE. This change is not permanent and will be lost when the window is closed.

Floating point numbers are displayed by default using the printf-style code %6.2f. To change this format for an EU conversion, the display format attribute is available for the EU definition.

Item Info
Right-clicking an item also allows the user to view the info page about that item, to get its description and other properties.

Changing Colors
Stale items are shown in the stale color on the display pages. This color is defaulted to purple, but can be set to a different value in a configuration file (Section ###).

Other color changes depend on the monitors assigned to that item and their trigger state. When the monitor is defined, the desired display page color can be specified. The item will then be that color whenever that monitor is in its triggered state. If there are multiple triggered monitors for an item, the color will be that of the last monitor in the list.

Creating Command Menus
Any command that can be typed at the HYDRA command prompt can be placed in a command menu. The contents of the menus are defined in XML files, which must reside in the Menus directory in the HYDRA run directory. Each file defines one menu. To change or add a menu, HYDRA must be restarted.

```
<isisMenu>
      <menu name="My Commands">
            <cmd name="command1"/>
            <cmd name="command2"/>
            <menu name="Nested Commands">
                  <cmd name="command3"/>
            </menu>
      </menu>
</isisMenu>
```

There is no limit to the number of sub menus and the level of nesting allowed.

Internal Commands FIXME reformat
The following commands are processed by HYDRA. All commands can be used on the command line, or in scripts, unless otherwise noted.

Item Access
Items defined in the HYDRA database can be accessed with the following commands.

print
This command prints the current value of the item named. The value printed will be converted if the item type is something other than DN.

>>print myItem

print_history
This command prints the last x values of the item, depending on its history setup. If there is no history defined, nothing is printed.

>>print_history myItem

set

The set commands allows the user to set the value of an item. The value will be overwritten if a frame containing the item arrives. The command also allows the value to be an expression, eg 1 + 1. If an expression, there must be spaces between the operands and the operator. The operands can be numbers or other items. Item values are converted to floating point before being operated on.

>>set myItem1 = 1
>>set myItem2 = myItem1
>>set myItem3 = myItem1 + myItem2

Only two operands are allowed, and the operators +, -, *, /, %, and ^.

verify

The verify command checks the value of the item against another item value or a number. If the check does not pass, HYDRA reports an error.

>>verify myItem1 == myItem2

The following operators are allowed: ==, !=, >, >=, <, <=, as well as && and || to group statements together. The entire thing must be all on one line (applicable to scripts mostly).

>>verify myItem == 1 && myVar == 2

To verify against a dictionary value, the following notation is used:

>>verify myItem == myDict(OFF)

export_item

Export item/frame to text file

>>export_item itemName fileName

Process Commands

The commands described in this section start and stop various HYDRA processes.

start

The start command is used to start the execution of a script.

>>start myScriptName.prc [engine="1,2,ALL"]

show_engine

This command shows a hidden script window for the provided engine number.

>>show_engine [ALL | engineList]

hide_engine
This command hides a script window for the provided engines.

>>hide_engine [ALL | engineList]

pause
This command pauses the execution of the provided engine.

>>pause [ALL | engineList]

step
This command puts the provided engine number into step mode, where one line is executed at a time.

>> step [ALL | engineList]

go
This command resumes execution of the provided engine.

>> go [ALL | engineList]

goto
This command jumps the provided engine to the line number or label number provided.

>>goto [engineList] label|line

parse
This command parses the provided script file, but does not execute it. It verifies the formatting of the file and will return success of failure.

>> parse scriptName

return_all
This command stops execution of all scripts within the given engine.

>> return_all [ALL | engineList]

return_wait
This command returns an engine out of the lowest sub script and pauses the script execution. If no sub script, the affect is the same as return_all.

>>return_wait [ALL | engineList]

return

This command returns an engine out of the lowest sub script and continues execution immediately. If no sub script, the affect is the same as return_all.

>> return [ALL | engineList]

FIXME create separate script section with go and return commands as well – move to other script section?

start_sender

This command starts a predefined sender. It has optional parameters if the command the sender uses takes parameters.

>>start_sender mySender param1 x param2 y

stop_sender

This command stops the execution of a sender.

>>stop_sender mySender

update_sender

Update a sender configuration

>>update_sender mySender interval myValue

start_reader

This command starts a reader, which will also open the hardware for the reader if it is not already open.

>>start_reader myReader

stop_reader

This command stops a reader execution. This will close the associated hardware.

>>stop_reader myReader

open

This command opens the requested hardware. The actual action depends on the type of hardware:

Server – no action
Client – attempts to connect to server
InFile – opens requested file – if no filename provided, the user is prompted
OutFile – no action
Serial – configures the serial port
OpalKelly – configures the FPGA

>>open myHardware

close

This command closes the requested hardware. As with open, the action taken depends on the type of hardware.

Server – no action
Client – disconnects from server
InFile – closes input file
OutFile – no action
Serial – no action
OpalKelly – no action

>>close myHardware

send_str

This command sends a string to the named hardware device. The string is converted into ascii bytes followed by a newline character before it is written to the hardware.

>>send_str myHardware Hello World

set_script_mode

Set the script engine execution mode.

>>set_script_mode [ALL| engineList…] MANUAL|AUTO|TEST

exit

Exit Hydra

>>exit

Configuration Commands

This section describes commands to change the current configuration of HYDRA.

create_config

Create a new config item.

set_frame_checksum

This command enables or disables the checksum calculation for the given frame. It has no affect if the frame has no checksum attribute.

>>set_frame_checksum frameName enable|disable

prot_cmd

This command enables or disables the protected command attribute. If disabled, no message box will appear when a protected command is sent.

>>prot_cmd enable|disable

enable_monitor
This command enables a monitor.

>> enable_monitor monitorName

disable_monitor
This command disables a monitor.

>> disable_monitor monitorName

monitor_clear
This command clears the triggered state of the provided monitor.

>> monitor_clear monitorName

monitor_cmd
This command sends a command to the monitor. See section FIXME for a list of possible commands.

>> monitor_cmd monitorName cmd

range_checking
This command enables or disables argument range checking for commands.

>>range_checking enable|disable

set_dec_header
This command changes the header frame for the decoder. See section FIXME for details on decoder headers.

>>set_det_header decoderName frameName|NONE

set_dec_output
This command changes the status of an output device for a decoder. See section FIXME for details on decoder outputs.

>> set_dec_output decoderName outDeviceName frameId|ALL enable|disable

cmd_dec
Custom command for decoder

>>cmd_dec decoderName cmdStr

set_cmd_dec
Set destination decoder for specified frame

>>set_cmd_dec frameName decoderName

save_xtce
Get LASP XTCE definition of command and telemetry database

>>save_xtce templateFile inputFile outputFile buildNumber

Other Commands
This section describes other commands for HYDRA.

noop
This command does no action except for printing that the command was received. It is most often used for testing a remote command connection.

create_page
This command creates a page definition for the provided frame. The definition will be saved in a file in the Pages directory with the frame name + .xml. If the frame does not exist, an empty definition is created. The page name is also added to the Display Pages menu so it can be viewed.

>>create_page myFrame

show_page
Thie command shows the desired display page.

>>show_page displayPage

create_config
The purpose of this command is to create a new database item without using a configuration file. The new item is not saved in a file, so it will only apply to the current instance of HYDRA.

>>create_config xmlNode attributes

example:
>>create_config itemDef name=myNewItem type=dn8

echo
This command echoes the provided string to the message window and log file. It is most commonly used in scripts.

>>echo Hello World!

note
This command writes the provided string to the runTimeNotes.txt file in the current run directory. The string will be prepended with the current time. See section FIXME for more detail on tags and notes.

>>note Test started

tag

This command writes the provided string to the rundirIndex.txt file in the top-level Rundirs directory. The string will be prepended with the current time. See section FIXME for more detail on tags and notes.

>> tag Test started

list_internal

This command prints a list of all of the internally defined items.

>>list_internal

load_config

This command causes HYDRA to read the provided file and create new database objects from that file. This is a useful command for creating large changes on the fly. Existing items are not changed.

>>load_config myConfigFile.xml

load_dc

This command loads a display definition file, which governs which windows are open and their layout. If no filename is provided, HYDRA will attempt to open the file default.dc.

>>load_dc myDisplayFile.dc

save_dc

This command saves the current window layout to a file, which can be loaded later with the load_dc command. If no filename is provided, the layout is saved to the file default.dc, overwriting any previous file.

>>save_dc myDisplayFile.dc

save_config

This command writes the current HYDRA configuration to the file config_out.xml.

>>save_config

show_count

This command shows the counts for every frame ID received by the requested decoder.

>>show_count myDecoder

show_frame
This command shows the detailed format of the provided frame.

>> show_frame frameName

show_dict
This command shows the details of the provided dictionary.

>> show_dict dictName

show_item
Show details of the item.

>>show_item itemName

rundir
This command provides access to the current run directory. There are two arguments for this command. The first creates a new run directory with the current timestamp. All new files will be written to the new directory. The second brings up a browser window for the current directory.

>>rundir new
>>rundir show

version
This command prints the current HYDRA version to the message window and event log.

>>version

update_pref
This command updates a configuration preference. See section FIXME for a list of the preferences.

>>update_pref prefName true|false

file
This command provides access to the file system. The first argument will copy a file to a new name, and the second will rename the provided file.

>>file copy oldName newName
>>file rename oldName newName

spawn
This command spawns a new process. It will run asynchronously to HYDRA in its own window.

>>spawn programName

shell

This command starts a new process, which will run inside HYDRA. Control will not return until the process completes. (Windows only)

>>shell programName

system

This command executes the provided system command.

>>system command

compare

This command compares two files and reports any differences. If there are differences, a file in the run directory will be created which details each difference line by line. The filename provided can be an absolute path, a file in the run directory, or the name of a HYDRA file (see section FIXME).

>>compare file1 file2

close_evtlog

This command closes the current event log and starts a new one with a new filename.

>>close_evtlog

plot

This command creates a real-time plot for the requested telemetry point.

>>plot itemName

plot_history

This command plots the history of a telemetry point. If not history is defined, this command has the same effect as plot.

>>plot_history itemName

start_seq

This command changes the command output stream to the sequence buffer. Commands will be sent to the buffer instead of the normal output until the stop_seq command is received. See Section FIXME for details on sequence mode.

>> start_seq

stop_seq

This command cancels the current sequence mode for command output. See section FIXME for details on sequence mode.

>> stop_seq filename headerFrame useCount

cmd_hw

This command sends a command to a hardware device. See section FIXME for a list of possible commands.

>>cmd_hw hwName cmdString

send_query

Send a query to hardware device

>>send_query queryName [noPrint]

cmd_query

Configure a specific query

>>cmd_query queryName cmdString

send_block

Send one iteration of a CmdBlock

>>send_block blockName

start_block

Start sending a command block at its period

>>start_block blockName

stop_block

Stop sending a command block

>>stop_block blockName

set_block

Set a parameter for a command block

>>set_block blockName parameter parameterValue

Debugging

debug

Debug for Beth

echo
Echo string to window.

>>echo stringToEcho

note
Write note to runTimeNotes.txt file

>>note noteToWrite

tag
Write to tag file (runDirIndex.txt). Tag file appears in Rundirs directory.

>>tag noteToWrite

rundir
Prints current directory, get a new directory, or open directory browser.

>>rundir [print|new|show]

version
Get current Hydra version information

>>version

Scripts
The HYDRA scripting language allows you to create repeatable tests and activities. The language has the following features:

- Flow-control (if statements, loops, gotos)
- Local variables
- Subscripts
- Wait statements (indefinite, timed, and conditional)

Script Engines
Each script is executed by a script engine. There are lots of engines, so multiple scripts can run concurrently. There can be many levels of nested scripts within a single engine, ie a script can call subscripts which return execution to the calling script when they complete. Scripts can also spawn new scripts to execute in a different engine.

A script engine executes commands in the script at a 10Hz rate. The user can control the engine by pausing, resuming, and jumping around within the executing

script. The engine window provides a view of the scripts that are running with the current line highlighted.

Whenever a new script is started, HYDRA finds the next available engine and the script begins execution. The engine number can be specified if certain engine numbers are assigned to specific purposes. All engines have the same capabilities. See Section ## for details on script commands.

Starting Scripts

Scripts are started by passing the name of the script file to the "start" command (Section ##). Script files must end in ".prc" but the extension is optional in the start command.

Script Directories

By default, HYDRA checks the base directory for script files when starting execution. If a script lives in a subdirectory, the full path can be specified in the start command. Another option is to setup a script directory hierarchy. This list of directories tells HYDRA where to look if the script is not found in the default location. The order of the directories matters if there are scripts with the same names in different locations. HYDRA will execute the first script it finds.

To setup a hierarchy, the scriptDir node is used in any of the config files.

```
<scriptDir name="Script"/>
<scriptDir name="Scripts/special"/>
```

In this example, the directory Script is checked first for the file, and then the special subdirectory. Absolute paths to scripts can also be provided if not a part of the HYDRA directory architecture.

Creating the hierarchy simplifies scripts because subscripts can be organized without the calling scripts needing to change. There can also be different versions of the same scripts, and the precedence can be updated by simply changing the order of the directories in the config file.

Script Control

There are two ways to control the execution of a running script: buttons in the script engine window, and command line commands. These commands are outlined in Section ###. They can control one or more engines at a time, while the buttons only apply to the current engine.

Parsing Scripts

Scripts are parsed each time they are called to construct a flow control tree and verify certain syntax. If the parse fails, the script cannot be run and an error is reported. The parse does not catch all syntax issues so there can still be run time errors. To parse a script without actually running it, the parse command is available.

>>parse myScript

Basic Script

A script is just a text file that contains commands to be executed by the engine. The commands are executed in the order they appear in the file, unless there is a flow control structure present, like a while loop or if statement. Any command that can be sent from the command line is legal inside of a script and will have the same affect. For instance, a command in a script with no parameters specified will still pop up an argument dialog if needed.

In a script, any text beginning with a ';' is considered to be a comment and is ignored by HYDRA. Comments should start at the beginning of a line.

Below is an example of a basic script:

File myNoop.prc

```
;This is my noop script
noop
noop
noop
```

Script execution begins with the following command:

>> start myNoop

The first line of the script is ignored and the three noops are sent a tenth of a second apart. After the last command, the script is complete and the engine returns to its IDLE state.

Wait Statements

There are several kinds of wait statements available to a script. The first is an indefinite wait. This wait pauses the script engine, and execution will not resume until the user provides a GO.

```
noop
noop
pause ;indefinite wait
noop
```

The second type of wait is a timed wait. This statement pauses execution for the specified amount of time.

```
noop
wait 1000 ; wait for 1000 ms
noop
```

This wait creates a 1 second gap between the two commands.

Then there are conditional waits. These statements pause execution until a condition is met. The conditions for waits (and all other conditionals mentioned in this section) are the same format as described for the verify command in Section ###.

```
Noop
tlmwait cmdCnt == 1 ; wait for the cmdCnt item to equal 1
Noop
```

The script will wait forever until cmdCnt is set to one. Since waiting for forever is not always desired, in case the condition is never true, HYDRA provides an optional timeout for these kinds of waits. See timeout.

```
Noop
tlmwait cmdCnt == 1 ? 5000 ; timeout after 5 seconds if condition not
met
```

If the timeout is reached, the script engine will stop the script and report an error. Related to this is the timeout wait. It does the opposite of a conditional wait, in that it reports an error if the condition passes and none of it times out.

```
tlmtimeout cmdCnt == 1 ? 5000 ; wait 5 seconds for count to not change
```

Another type of wait is the absolute wait. It pauses execution until a specific time. The time must be in the format YYYY/DOY-HH:MM:SS.

```
abswait 2015/206-12:00:00
```

Flow Control

Flow control statements do not take anytime to execute, ie they are not counted in the 10Hz execution time of a script.

If-Statements

An if-statement causes the script to branch and execute different commands based on a condition.

```
Noop
if cmdCnt == 1
  Noop
  Noop
else
  Noop
endif
```

In this example, if cmdCnt is 1, then two noops are sent, otherwise, just one noop is sent.

If-statements can be nested inside each other. The only required statement is the first if, the else and any elif's are optional. The endif is also required.

```
if cmdCnt == 2
      if tlmCnt == 7
            echo 3
      else
            echo 4
      endif
elif cmdCnt == 1
      echo 1
else
      echo 2
endif
```

Numbered Loops

A numbered loop is used to repeat the same commands a fixed number of times.

```
loop 10
      Noop
endloop
```

For this example, 10 noops are sent.

Foreach Loops

Foreach loops iterate through a list of items, assigning a local variable to a different value each iteration.

```
foreach channel in "1,2,3"
      print channel
endfor
```

While Loops

While loops repeat until the provided condition evaluates to false.

```
while cmdCnt < 10
  Noop
endwhile
```

Other Flow Control Statements

Like in other scripting languages, the keywords break and continue are also available to control the execution of a loop. Break will immediately exit the loop, even if there are more iterations remaining. Continue will immediately return to the top of the loop, skipping any statements that may succeed it.

Labels and GOTO

A different type of flow-control statement is the GOTO command. This command causes the script to jump to a different line in the script, either by line number or by label reference.

```
CASE1:

Noop
Noop
goto CASE3

CASE2:
Noop

CASE3:
Noop
Noop
```

Any statement ending in a ':' is considered by HYDRA to be a label and can be used in a GOTO command. In this example, the noop in CASE2 is skipped over as execution jumps to the CASE3 label instead. GOTOs can jump anywhere in an existing script, but should not be used to enter or exit a loop. In that case, the loop conditions will not be initialized correctly and the behavior is undefined. Within a loop, goto's are ok.

GOTOs are often used in conjunction with if-statements.

```
if instrMode == 1
  GOTO CASE1
else
  GOTO CASE2
endif

CASE1:
Noop
GOTO END

CASE2:
Noop
Noop
GOTO END

END:
;end of script
```

Variables

A script can declare local variables for use inside the script only. These values are not accessible to the main HYDRA program, or any subscripts called by the script.

To create a variable, the variable name and type must be provided in the 'declare' command.

```
declare myVar1 dn8
declare myVar2 float32
```

Once declared, the variables can be used just as other items defined in the main HYDRA program. If there are conflicting names, the local variable takes precedence over the global.

When using variables as command arguments or in verify statements, they can be referenced with the $ symbol. This causes HYDRA to substitute the value of the variable in place of its name before executing the line.

```
declare myVar dn8
set myVar = 1

send_command arg1 $myVar

tlmwait cmdCnt == $myVar
```

The $ notation can only be used with local variables, not global items or telemetry.

Variables can be declared anywhere in the script, but must be declared before they are referenced. Declaring the same variable twice results in an error. Variable declarations are not considered commands, so there is no wait between the statements.

The script can prompt the user for a variable value using the ask command.

```
ask "What value for myArg?" myArg
```

This will pop up a message box to prompt for the value. If only certain values are allowed, those can be supplied to the command as well.

```
ask "Which value for myArg?" myArg "1,2,3"
```

String Variables

A special type is available only inside of scripts, called varString. This type declares a variable that holds a string value, rather than an integer or a double. It can be set and dereferenced, but should not be used in verify statements.

```
declare myStr varString
set myStr = ch1

tlmwait $myStrState == 0
```

In this example, the string is set to "ch1" and then that value is combined with state to create the variable name "ch1State", which is then checked against a value. These strings can also be passed as arguments.

Arguments

Arguments are like variables, except their initial values are set when the script is called. Otherwise, they are used exactly the same. The arguments must be declared in the same order in which their values will be supplied.

```
argument myArg1 dn8
argument myArg2 varString
```

This script is called using parentheses to set the argument values.

```
start myScript(1, ch1)
```

Subscripts

A subscript is a script that executes in the same engine as the calling script. The main script is halted until the subscript completes. There is no limit to the number of nested scripts. To identify the new script as a subscript, instead of a script to start in a new engine, the 'call' command is used.

```
call mySubscript(1, ch1)
```

Return Statements

Once the script engine has processed the last line in the file, the script execution is complete and the engine returns to IDLE. To stop a script before that case, the 'return' statement is used. If the script is a subscript, 'return wait' will cause the higher level script to pause and wait for a go before continuing. 'return all' cancels all scripts in the engine.

Script Execution

While running, the user can control the script execution to override the controls within the script itself. For example, the user can pause the script at any time and issue go's to move through timed waits or errors. The user can also jump around in the script with GOTO commands to repeat or skip certain sections. They can stop script execution with the 'return' statements discussed above. There are controls for all these functions in the script engine window. FIXME step command

Testing

The Test framework is designed to be used from scripts, but the commands can be sent from the command line as well. When running a test case, all Hydra artifacts produced will be stored in a directory specific to the test that is being run. This includes any event logs or telemetry files generated during the test. The path for the artifacts is:

basePath/testCSCI/testGroup/testName/date
where:

- basePath – provided in testing configuration
- testCSCI – optional
- group – provided when test starts
- name – provided when test starts
- date – start time of the test

All paths are relative to the Hydra project directory.

Each test case run is also logged to a result file, in csv format for later viewing and modifying with Excel or other spreadsheet program. The result file is appended for each test case run and contains columns for the test csci, group, name, start time, end time, hydra version, script version, notes, host, test conductor, etc. The name and location of the result file is provided in the testing configuration.

Hydra can create a summary of all test cases in its project directory and output the results to another csv file, with the columns in the following order: test csci, group, name, script name, requirements, author, description, prerequisites, notes, etc. The location and name of this file is provided when the test cases command is executed.

A report can be created at the end of a set of tests with the **test groups** command. The default name and location for the output file is provided in the testing configuration, but can be overridden when the command is executed. The format of this report is XML and matches existing formats for unit tests for runners like Google Test and JUnit.

The test framework can be completely disabled, in which case Hydra ignores all of the test commands.

View Test Commands

To view all available test commands:

test help

This displays a list of all available actions. To see usage information for a specific action, type:

test help action

Return Values

Scripts can return values to their callers via the **return** command. The caller can access the value returned with the special variable **$$ret**. Floating point values that are returned will be truncated to the format %6.6f.

Script Modes

Hydra provides a new mode for running scripts, called AUTO. The old behavior is now MANUAL mode, which is the default. In AUTO mode, pause statements have no

effect, verify statements do not pause the script, and ask statements use the default provided instead of prompting the user. Auto mode does not disable protected command checking, so that needs to be disabled separately, if desired.

The design of AUTO mode is to limit user interaction with a script so tests can be performed in an automated fashion.

Script errors will still pause execution, such as invalid telemetry item name, invalid command, badly formatted if statement.

Ask Statements

Ask statements are used to get input from the user. They have two forms, choices and freeform input. A default value should be provided for the answer in case the script engine is set to AUTO mode.

For a freeform *ask* statement, the format is:

ask "What is up?" nothing

where **nothing** is the default answer in AUTO mode. In this form, the user can enter whatever value they want for the answer.

For a *choices* ask statement, the format is:

Ask "What is up?" @everything,nothing,theSky

where **everything** is the default answer in AUTO mode. This form presents the user with a list of choices for the answer and they must pick from the choice.

Testing Configuration

The testing framework must be configured via Hydra config file before it can be used. Below is an example of a configuration:

```
<testing>
        <archiveDirectory path="test"/>
        <resultFile name="test/test_results.csv"/>
        <reportFile name="test/test_report.xml"/>

        <version cmd="bash git_version.sh"/>
        <setup>
                <script name="testSetup"/>
        </setup>

        <teardown>
                <script name="testTeardown"/>
        </teardown>
```

</testing>

All configuration nodes must be inside a `<testing>` node. This node can exist anywhere in the hierarchy of Hydra config files. Multiple nodes can be used, and duplicates will override the previous entry.

<archiveDirectory>	Specifies path to the artifacts directory, relative to the Hydra project
<resultFile>	Default name for result file
<reportFile>	Default name for report file
<version>	Provides command used to get the version information for the script being run (svn, git, etc). This command is run on the shell and the output is captured and saved. A shell script can be provided if a single command is insufficient.
<setup> *optional*	Name of the script to run at test start
<teardown> *optional*	Name of the script to run at test end

Test Cases

A test case is just a Hydra script with some special keywords to facilitate testing. The test case script can have all commands and checks within in, or it can call other scripts.

Required Commands:
test start name="testName" group="testGroup" [csci="testCSCI"]
test end

Optional Commands:
test summary
test fail [message]
test requirement reqNumber
test author Name
test description Description
test notes Notes
test prereq Prerequisite

If **csci** is provided, the path for the test results will include the csci directory.

Upon **test start**, any script provided in the testing config for setup will be executed.

Upon **test end**, the original Hydra run directory is restored, and the teardown script provided in the testing configuration will be executed.

The **test fail** command will log test failures to the Hydra event log and the test report.

The **test note** command adds a note to the event log and to the result file. These can be typed by the user at the command line to add notes to a running test that may have an error or special activity.

The **test summary** command prints to the event log a summary of the test run, including the number of failures and if the test passes (no failures recorded).

The other test optional commands are used to generate the test cases summary described in the overview.

Test Report

Require Commands:
test reset
test groups

The **test reset** command clears the Hydra memory of previous tests, and should be provided before any suite of tests is run.

The **test groups** command generates the report for all test cases run since the last reset.

Other Commands

Test Conductor

test conductor sets the name of the current test conductor, for use with the result file. If no conductor is provided, the current user name is used instead.

Testing State

test state sets the state of the testing framework to on or off.

Setup/Teardown Scripts

test setup/teardown changes the name of the setup/teardown script from the one provided in the testing configuration.

Telemetry Verification

The **verify** command is a Hydra command that reports an error if an expression does not evaluate to true:

verify cmdCount == 1

If testing is enabled, if a verify statement fails it will automatically log the failure as test fail for the test report. It can be used as a shortcut for the following:

```
if cmdCount != 1
        test fail Bad command count
endif
```

In MANUAL script mode, this error will cause the script to pause. In AUTO mode, the script will continue past this error, so if action needs to be taken, the if statement is a better construct to use.

See also verify

Telemetry Wait Timeouts

The **timeout** command is used to catch a telemetry timeout so action can be taken.

```
tlmwait cmdCount == 1 ? 4000
timeout
   test fail cmdCount not == 1
endtimeout
```

This example waits 4 seconds for the condition to be true.

See also tlmwait

This feature is available in 1.9.22306 or higher.

Verify with Tolerance

A special operator is availble to check equality with tolerance.

verify temperature ~= 30.0 : 1.0

This command returns true if the temperature is 30.0 +/- 1.0.

This feature is available in 1.9.22306 or higher.

Examples

Examples of a test case script and test driver script shown below:

;;;;;Test Case;;;;;;;;;

test start name="myTest" group="testGroup" csci="ram"

test author Dr Suess
test requirement 4.2
test description An example test case

```
test notes Some more info about this test case
test prereq none

BEGIN:
call sendNoopCmd

tlmwait noop_count == 1 ? 5000

timeout
        test fail Noop command not accepted
        goto FINISH
endtimeout

call sendNoopCmd

if noop_count == 1
   test fail Noop command not accepted
   goto FINISH
endif

FINISH:
test summary
test end

;;;;;;;;;;;;;;;;;;;;;;;

;;;;Test Driver;;;;;;;;

test state on
test reset

test conductor George Washington

call unitTest1
call unitTest2
call unitTest3

FINISH:
test groups

;;;;;;;;;;;;;;;;;;;;;;;
```

XML Reference

This section contains details about the xml nodes used in the HYDRA configuration files. See Section ### for context on how to apply these elements to a specific project.

Config Files

<color>

Description

Sets a new value for an HYDRA color.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the color to change	YES
code	string	Name or rgb code for the new color	YES

Example

```
<color name="KEYBOARD" code="PURPLE"/>
```

<dictionary>

Description

Defines the mapping between integer keys and strings.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Unique name used to identify the dictionary	YES

Children

<pair>
Occurrences: [0 … *]
Description: Maps integer to string
Attributes:

Name	Type	Description	Required
key	integer	Value to associate the string with	YES
str	string	String for this key – can have up to four format codes	YES

Example

```
<dictDef name="myDictionary">
      <pair key="1" str="ON"/>
      <pair key="2" str="OFF"/>
</dictDef>
```

<euPoly>

Description
Defines a polynomial for conversion from DN to engineering units.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Unique name used to identify the conversion	YES
category	string	Type of conversion – if not specified, the default is POLY	NO

Children

<coeff>
Occurrences: [0 … *]
Description: Maps coefficient to power of x
Attributes:

Name	Type	Description	Required
index	integer	Power of x for this coefficient	YES
value	float	Coefficient value	YES

Example

```
<euDef name="myEU">
      <coeff index="0" value="2.9"/>
      <coeff index="1" value=".003"/>
</euDef>
```

<euEXIS>

Description
Declares an EXIS temperature conversion.

Parent
<hydraDef>

Example

```
<euEXIS name="myEXISTemp"/>
```

<include>

Description
Contains the name of another configuration file for HYDRA to process

Parent
<hydraDef>

Example

```
<include>myFile.xml</include>
```

<typeDN>

Description
Declares a data number type.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
size	integer	Number of bits for the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO

Example

```
<typeDN name="myType" size="16" endian="LITTLE"/>
```

<typeSN>

Description
Declares a signed data number type.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
size	integer	Number of bits for the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO

Example

```
<typeSN name="myType" size="16" endian="BIG"/>
```

<typeState>

Description
Declares a state conversion type that uses a dictionary to convert integers to strings.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
size	integer	Number of bits for the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO
dict	string	Name of dictionary to use for conversion	YES

Example

```
<typeState name="myType" size="8" endian="LITTLE"
dict="myDictionary"/>
```

<typeEU>

Description
Declares an engineering unit type to convert integers to floating point.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
size	integer	Number of bits for the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO
conversion	string	Name of EU conversion to use	YES

Example

```
<typeEU name="myType" size="16" endian="BIG" conversion="myEU"/>
```

<typeFloat>

Description
Declares a floating-point number type. This type is always 32 bits.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO

Example

```
<typeFloat name="myType" endian="BIG"/>
```

<typeDouble>

Description

Declares a double precision floating-point number type. This type is always 64 bits.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO

Example

```
<typeDouble name="myType" endian="BIG"/>
```

<typeChar>

Description

Declares a character type. This type is always 8 bits.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the type	YES
endian	string	Indicates endianness of item (BIG*, LITTLE)	NO

Example

```
<typeChar name="myType" endian="LITTLE"/>
```

<itemDef>

Parent

<hydraDef>
<frameDef>

Description

Declares an item that will be used to store a value within HYDRA.

Attributes

Name	Type	Description	Required
name	string	Unique name for the item	YES
type	string	Name of the type for this item	YES
value	string	Initial value for this item, array format is “0 1 2 3” (default 0)	NO
num	integer	Number of elements for this item (default 1)	NO
history	integer	Number of samples to save for this item (default 0)	NO

Children

<description>

Example

```
<itemDef name=”myItem” type=”dn8” num=”3” value=”1 2 3”
history=”20”/>
```

<itemCounter>

Description

Declares a counter that will update when added to a frame.

Parent

<hydraDef>
<frameDef>

Attributes

Name	Type	Description	Required
name	string	Unique name for the item	YES
type	string	Name of the type for this item	YES
value	string	Initial value for this item, array format is “0 1 2 3” (default 0)	NO
num	integer	Number of elements for this item (default 1)	NO
history	integer	Number of samples to save for this item (default 0)	NO
incr	integer	Amount to increment counter	YES
max	integer	Maximum value allowed for counter	YES
min	integer	Minimum value allowed for counter	YES

Children

<description>

Example

```
<itemCounter name=”myItem” type=”dn8” value=”1” incr=”1” min=”1”
max=”10”/>
```

<itemDerived>

Description
Declares a derived item that takes its value from other items

Parent
<hydraDef>
<frameDef>

Attributes

Name	Type	Description	Required
name	string	Unique name for the item	YES
type	string	Name of the type for this item	YES
value	string	Initial value for this item, array format is “0 1 2 3” (default 0)	NO
num	integer	Number of elements for this item (default 1)	NO
history	integer	Number of samples to save for this item (default 0)	NO
fact1	string	Name of item for first factor	YES
fact2	string	Name of item for second factor, or floating point number	YES
opcode	string	Opcode for arithmetic operation (+, -, *, /)	YES

Children
<description>

Example
```
<itemDerived name=”myItem” type=”dn8” fact1=”myItem1” fact2=”3.2”
opcode=”*”/>
```

<itemTime>

Description
Declares a time item that computes its own value whenever it is accessed

Parent
<hydraDef>
<frameDef>

Attributes

Name	Type	Description	Required
name	string	Unique name for the item	YES
type	string	Name of the type for this item	YES
value	string	Initial value for this item, array format is “0 1 2 3” (default 0)	NO
num	integer	Number of elements for this item (default 1)	NO
method	string	Method to use for time calculation (secSinceEpoch, msSinceMidnight, daySinceEpoch)	YES

epoch	string	Epoch to use for time calculation (e.g. Jan 1 2000 00:00:00)	YES

Children
<description>

Example
```
<itemTime name="myTime" type="dn32" method="secSinceEpoch"
epoch="Jan 1 2000 00:00:00"/>
```

<argument>

Description
Declares an argument whose value must be provided by the user when adding to a frame. Must be used inside of a frame definition.

Parent
<frameDef>

Attributes
Name	Type	Description	Required
id	string	Name of this argument	YES
type	string	Name of the type for this item	YES
num	integer	Number of elements for this item (default 1)	NO
default	string	Default value for item if not specified	NO
min	integer	Minimum allowed value for item	NO
minCheck	string	Indicates if minimum value should be checked (enabled/disabled)	NO
max	integer	Maximum allowed value for item	NO
maxCheck	string	Indicates is maximum value should be checked (enabled/disabled)	NO

Children
<description>

Example
```
<argument id="group" type="dn8" default="1" min="1"
minCheck="enabled" max="10" maxCheck="enabled"/>
```

<frameDef>

Description
Defines an arrangement of items within a frame of bytes.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Unique name used to identify the frame	YES
id	integer	ID of the frame, used to identify frames within incoming blocks	NO
history	integer	Number of samples to save for all items contained in this frame	NO
decoder	string	Name of decoder that will build this frame	NO

Children

<field>

<archive>

Occurrences: [0 … 1]
Describes: Defines how to archive frame to file.
Attributes

Name	Type	Description	Required
prefix	string	File prefix for storage	YES
maxSize	integer	Max file size before creating a new one	NO
enabled	bool	Indicates if archiving is enabled, default is false	NO
ert	string	Indicates if ERT timestamps are enabled, default is disabled	NO
timestamp	string	Name of item to use as timestamp	NO

<stale>

Occurrences: [0 … 1]
Describes: Defines maximum allowed interval between frames
Attributes:

Name	Type	Description	Required
interval	integer	Number of ms to wait for Frame	YES
enabled	bool	Indicates if stale checking is enabled (true or false), default is false	NO
notify	string	Name of notification to use if condition triggers	NO

<pad>

Occurrences: [0 … 1]
Description: Defines padding requirements for constructing frame
Attributes:

Name	Type	Description	Required
bytes	integer	Multiple of bytes to pad this frame to	NO*
bits	integer	Multiple of bits to pad this frame to	NO*
value	integer	Value to use for padding – default 0	NO

<protected>

Occurrences: [0 … 1]

Description: Indicates protected status for frame
Attributes:

Name	Type	Description	Required
enabled	bool	Indicates if protected status is enabled or not	YES
msg	string	Message to appear in confirmation box	NO

<header>
Occurrences: [0 … 1]
Description: Specifies header frame for this frame
Attributes:

Name	Type	Description	Required
frame	string	Name of the header frame that contains the <placeholder> tag	YES

<checksum>
Occurrences: [0 … *]
Description: Defines a checksum to perform over members of this frame
Attributes:

Name	Type	Description	Required
item	string	Name of item within frame to place the checksum result	YES
seed	integer	Starting value of checksum calculation	YES
start	string	Name of item within frame to begin calculation	YES
stop	string	Name of item within frame to end calculation – or END	YES
method	string	Calculation method for checksum (xor, ip, udp, crc)	YES

Method CRC Additional Attributes

Attribute	Description	Required
poly	Polynomial to use in CRC calculation	YES
width	Size of the crc	YES

Method UDP Additional Attributes

Attribute	Description	Required
phLen	Location of length for pseudo-header	YES
phPro	Location of protocol for pseudo-header	YES
phSrc	Location of source for pseudo-header	YES
phDst	Location of destination for pseudo-header	YES

<length>
Occurrences: [0 … *]
Description: Defines a length to perform over members of this frame
Attributes:

Attribute	Description	Required

item	Name of item within frame to place the length result	YES
offset	Offset (in bits) to apply to length (default 0)	NO

Example

```
<frameDef name="myFrame" id="0x02" decoder="myDecoder">
      <stale interval="10000" enabled="true"/>
      <pad bits="32" value="0"/>
      <field>
            <itemDef name="myItem" type="dn8"/>
      </field>
</frameDef>
```

<field>

Description
Defines the arrangement of items within a frame definition.

Parent
<frameDef>

Attributes

Name	type	Description	Required
startByte	integer	Indicates start position of items within the tag (default 0)	NO
startBit	integer	Indicates start position of items within the tag (default 0)	NO

Children
<itemDef>

<item>
Occurrences: [0 … *]
Description: Indicates which item should be in this location
Attributes:

Name	Type	Description	Required
name	string	Name of item at this frame position	YES

<itemCopy>
Occurrences: [0 … *]
Description: Creates a copy of a previously declared item
Attributes:

Name	Type	Description	Required
name	string	Name of item to copy	YES
suffix	string	Suffix to add to end of item name to identify the copy	YES

<opcode>
Occurrences: [0 … 1]
Description: Defines a special field called the opcode

Attributes:

Name	Type	Description	Required
value	integer	Value of opcode	YES
type	string	Name of the type for this opcode	YES

<argument>

<itemCounter>

<itemTime>

<itemDerived>

<placeholder>
Occurrences: [0 … 1]
Description: Indicates location where frames should be placed, if this frame is a header
Attributes:

Name	Type	Description	Required
size	integer	Size of the placeholder, if known	NO

Example

```
<frameDef name="myFrame">
      <field startByte="0" startBit="0">
            <item name="myItem1/>
            <item name="myItem2"/>
      </field>

      <field startByte="24" startBit="0">
            <itemCopy name="myItem1" suffix="_v2"/>
            <itemCopy name="myItem2" suffix="_v2"/>
            <item name="myItem"/>
            <placeholder size="256"/>
      </field>
</frameDef>
```

<command>
See <frameDef>.

<alias>

Description
Declares a command alias.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required

name	string	Name of the alias	YES
command	string	Name of the command to alias	YES

Children

<argument>
Occurrences: [0 … *]
Description: Provides new value for argument id
Attributes:

Name	Type	Description	Required
id	string	ID of the argument from the command	YES
value	string	Value of the argument	YES

Example

```
<alias name="myCommandAlias" item="myCommand">
      <argument id="arg1" value="value1"/>
      <argument id="arg2" value="value2"/>
</alias>
```

<senderDef>

Description

Defines a sender process to send commands at regular intervals.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Unique name for this sender	YES
interval	integer	Number of ms between each frame	YES
frame	string	Name of the frame to send	YES

Example

```
<senderDef name="mySender" interval="2000" frame="noop"/>
```

<senderFile>

Description

Defines a sender process to send memory load commands.

Attributes

Name	Type	Description	Required
name	string	Unique name for this sender	YES
interval	integer	Number of ms between each frame	YES
frame	string	Name of the frame to send	YES

fileType	string	Type of file to send (bin, hex)	YES
length	string	Name of field within frame to set length of chunk	NO
addr	string	Name of field within frame to set address of chunk	NO
buffer	string	Name of field within frame to populate with buffer contents	NO

Children

<defaults>
Occurrences: [0 … *]
Description: Defines default values for sender parameters
Attributes:

Name	Type	Description	Required
filename	string	Name of file to send	NO
start	integer	Start address for frame	NO
total	integer	Total number of bytes from file to send	NO
offset	integer	Offset within file to start reading from	NO
chunk	integer	Size of file chunk to send	NO

Example

```
<senderFile name="myFileSender" interval="1000" frame="memLoad"
fileType="bin" buffer="memBuffer" addr="memAddr"
length="memLength">
      <defaults filename="myFile.bin" start="0" total="65536"
offset="0" chunk="128"/>
</senderFile>
```

<monitorUpdate>

Description

Declares a monitor that triggers whenever its item updates.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO
color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO

silent	bool	Indicates if notifications and messages should be sent	NO

Children
<notify>

Example

```
<monitorUpdate item="myItem" enabled="true" color="blue"
msg="red" silent="false"/>
```

<monitorChange>

Description

Attributes

Name	Type	Description	Required
item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO
color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO
silent	bool	Indicates if notifications and messages should be sent	NO

Children
<notify>

Example

```
<monitorChange item="myItem" enabled="true"
condition="myCondition" color="rgb(0,0,255)" msg="black"/>
```

<monitorNoChange>

Description

Attributes

Name	Type	Description	Required
item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO

color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO
silent	bool	Indicates if notifications and messages should be sent	NO

Children
<notify>

Example

```
<monitorNoChange item="myItem" enabled="true"
color="rgb(0,0,255)" msg="black"/>
```

<monitorState>

Description

Attributes

Name	Type	Description	Required
item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO
color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO
silent	bool	Indicates if notifications and messages should be sent	NO
state	string	State to check against – for STATE monitors	YES

Children
<notify>

Example

```
<monitorState item="myState" enabled="true" color="rgb(0,0,255)"
msg="black" state="ON"/>
```

<monitorValue>

Description

Attributes

Name	Type	Description	Required

item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO
color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO
silent	bool	Indicates if notifications and messages should be sent	NO
operator	string	Operation to use to check value (==, !=, >, <)	YES
value	float	Value to check against	YES

Children

<notify>

Example

```
<monitorValue item="myItem" enabled="true" color="rgb(0,0,255)"
msg="black" operator="==" value="5"/>
```

<monitorRange>

Description

Attributes

Name	Type	Description	Required
item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO**
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO
color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO
silent	bool	Indicates if notifications and messages should be sent	NO
bound	string	Indicates if should check inside or outside of range	YES
low	float	Low value to check	YES
high	float	High value to check	YES

Children

<notify>

Example

```
<monitorRange item="myItem" enabled="true"
condition="myCondition" color="rgb(0,0,255)" msg="black"
bound="inside" low="0" high="10"/>
```

<monitorDelta>

Description

Attributes

Name	Type	Description	Required
item	string	Name of item to monitor	YES
name	string	Name of the monitor	NO**
enabled	bool	Indicates is monitoring is enabled	NO
condition	string	Name of another monitor – condition must be triggered before this monitor can trigger	NO
color	string	If color is specified, the Item will be changed color on the screen while the condition is triggered	NO
msg	string	Indicates the color of the message to print when the condition is triggered	NO
silent	bool	Indicates if notifications and messages should be sent	NO
delta	float	Delta to check against	YES

Children

<notify>

Example

```
<monitorDelta item="myItem" enabled="true"
condition="myCondition" color="rgb(0,0,255)" msg="black"
delta="1"/>
```

<notifySound>

Description

Defines a notification to send for a monitor or stale trigger.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of notification	YES
file	string	Name of .wav file to play when triggered	YES

Example

```
<notifySound name="soundNotify" file="mySound.wav"/>
```

<notifyEmail>

Description

Defines a notification to send for a monitor or stale trigger.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of notification	YES
addr	string	Email address to send notification to	YES

Example

```
<notifyEmail name="myEmail" addr="myEmail@email.com"/>
```

<msgDef>

Description

Defines the contents of an event message.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of event message	YES
frame	string	Frame to associate with message	YES
mainWindow	bool	Indicates if messages should be displayed on main HYDRA window	NO

Children

<archiveMsg>

Occurrences: [0 … 1]

Describes: Defines how to archive messages to file

Attributes

Name	Type	Description	Required
prefix	string	File prefix for storage	YES
maxSize	integer	Max file size before creating a new one	NO
enabled	bool	Indicates if archiving is enabled, default is false	NO

<item>

Name	Type	Description	Required
name	string	Name of the item	YES

format	string	printf style format string	NO
conversion	string	Name of dictionary to use	NO
param0	string	Name of item to use as first parameter for conversion	NO
param1	string	Name of item to use as second parameter for conversion	NO
param2	string	Name of item to use as third parameter for conversion	NO
param3	string	Name of item to use as fourth parameter for conversion	NO

<const>

Name	Type	Description	Required
str	string	Contents of string	YES

Example

```
<msgDef name="myMsg" frame="myFrame" mainWindow="true">
      <const str="MSG: "/>
      <item name="myItem" conversion="myDictionary"
param0="paramItem0" param1="paramItem1"/>
</msgDef>
```

<readerFixed>

Description

Defines a reader to gather frames from a data stream.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the reader	YES
hw	string	Hardware device to read from	YES
decoder	string	Decoder to send bytes to	YES
size	integer	Size of initial device read	YES
autoStart	bool	Indicates reader should start at system initialization	NO

Example

```
<readerFixed name="myReader" hw="myHW" decoder="myDec"
size="1024" autoStart="false"/>
```

<readerHeader>

Description

Attributes

Name	Type	Description	Required
name	string	Name of the reader	YES
hw	string	Hardware device to read from	YES
decoder	string	Decoder to send frames to	YES
lengthField	string	Indicates location and type of length within header	YES
headerSize	integer	Size of header	YES
maxSize	integer	Max size of packet that can be read from device	YES
offset	integer	Offset to apply to length – default is zero	NO
autoStart	bool	Indicates reader should start at initialization	NO

Example

```
<readerHeader name="myReader" hw="myHW" decoder="myDec"
lengthField="0:0:dn8" headerSize="2" maxSize="1024" offset="1"
autoStart="true"/>
```

<hwInFile>

Description

Defines an input file interface.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the hardware	YES
filename	string	Name of file to read	NO
interval	integer	Interval in ms between file reads	YES

Example

```
<hwInFile name="myFile" filename="infile.bin" interval="1000"/>
```

<hwOutFile>

Description

Defines an output file interface.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the hardware	YES
filename	string	Name of file to read	NO

Example

```
<hwOutFile name="myFile" filename="outfile.bin"/>
```

<hwServer>

Description

Defines server socket interface.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the interface	YES
port	integer	Port of the socket	YES
mode	string	Indicates read or write mode for the socket	YES

Example

```
<hwServer name="myServer" port="2000" mode="read"/>
```

<hwClient>

Description

Defines a client socket interface.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the interface	YES
port	integer	Port of the host	YES
addr	string	Address of the host	YES
mode	string	Indicates read or write mode for the socket	YES

Example

```
<hwClient name="myClient" port="2001" addr="localhost"
mode="write"/>
```

<hwSerial>

Description
Defines a serial interface.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the interface	YES
port	string	Name of the serial port	YES
baud	integer	Baud rate	YES
parity	string	EVEN, ODD or NONE parity	YES
stopbits	integer	Number of stop bits (1 or 2)	YES

Example

```
<hwSerial name="mySerial" port="COM1" baud="4800" parity="EVEN"
stopbits="1"/>
```

<hwOpalKelly>

Description
Defines an opal Kelly interface.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the interface	YES
configFile	string	Name of the FPGA configuration file	YES

Example

```
<hwOpalKelly name="myOK" configFile="myConfig.bit"/>
```

<decoderDef>

Description
Defines a decoder to process frames.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES

updateDB	bool	Indicates if this decoder should update the DB with the values from the frame def	NO
archive	bool	Indicates if this decoder should archive the frames received	NO

Children
<frameID>
<frame>
<outputDevice>
<outBuffer>

Example

```
<decoderDef name="myDecoder" updateDB="true" archive="true"/>
```

<decoderBCH>

Description
Defines a decoder to process frames and add BCH codes to the frame.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES
updateDB	bool	Indicates if this decoder should update the DB with the values from the frame def	NO
archive	bool	Indicates if this decoder should archive the frames received	NO
blockSize	integer	Size of BCH blocks	YES

Children
<frameID>
<frame>
<outputDevice>
<outBuffer>

Example

```
<decoderBCH name="myDecoder" updateDB="true" archive="true"
blockSize="8"/>
```

<decoderHdr>

Description
Defines a decoder to process frames after removing a fixed header.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES
updateDB	bool	Indicates if this decoder should update the DB with the values from the frame def	NO
archive	bool	Indicates if this decoder should archive the frames received	NO
length	integer	Size of header to remove	YES

Children
<frameID>
<frame>
<outputDevice>
<outBuffer>

Example

```
<decoderHdr name="myDecoder" updateDB="true" archive="true"
length="4"/>
```

<decoderSub>

Description
Defines a decoder to process frames and extract sub packets.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES
updateDB	bool	Indicates if this decoder should update the DB with the values from the frame def	NO
archive	bool	Indicates if this decoder should archive the frames received	NO
startByte	integer	Position within frame where subpackets begin	YES
lengthOffset	integer	Offset to apply to length field to get subpacket length	NO
lengthField	string	Indicates where within sub packet the length can be found	YES

Children
<frameID>
<frame>

<outputDevice>
<outBuffer>

Example

```
<decoderSub name="myDecoder" updateDB="true" archive="true"
startByte="4" lengthOffset="1" lengthField="myFrame.length"/>
```

<decoderVCDU>

Description

Defines a decoder to process VCDU frames and extract subpackets.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES
updateDB	bool	Indicates if this decoder should update the DB with the values from the frame def	NO
archive	bool	Indicates if this decoder should archive the frames received	NO
startByte	string	Position within frame where subpackets begin	YES
lengthOffset	string	Offset to apply to length field to get subpacket length	NO
lengthField	string	Indicates where within sub packet the length can be found	YES
firstHeaderField	string	Indicates where first header pointer field can be found	YES
sequenceField	string	Indicates where sequence field is in frame	YES

Children

<frameID>
<frame>
<outputDevice>
<outBuffer>

Example

```
<decoderVCDU name="myDecoder" updateDB="true" archive="true"
startByte="4" lengthOffset="1" lengthField="0:0:dn8"
firstHeaderField="myFrame.firstHeader"
sequenceField="vcduFrame.sequence"/>
```

<decoderStr>

Description
Defines a decoder to turn a frame into a string.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES

Children
<frameID>
<frame>
<outputDevice>
<outBuffer>

Example

```
<decoderStr name="myStrDecoder"/>
```

<decoderCmd>

Description
Defines a decoder to turn a frame into an HYDRA command.

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES

Children
<frameID>
<frame>
<outputDevice>
<outBuffer>

Example

```
<decoderCmd name="myCmdDecoder"/>
```

<decoderyQuery>

Parent
<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES

Children

Example

```
<decoderQuery name="myDecoder"/>
```

<frameID>

Parent

<decoderDef>
<decoderBCH>
<decoderVCDU>
<decoderSub>
<decoderHdr>
<decoderStr>
<decoderCmd>

Attributes

Name	Type	Description	Required
name	string	Location of the id within the frame	YES

Example

```
<decoderDef name="myDecoder">
      <frameID name="0:0:dn8"/>
</decoderDef>
```

<frame>

Parent

<decoderDef>
<decoderBCH>
<decoderVCDU>
<decoderSub>
<decoderHdr>
<decoderStr>
<decoderCmd>

Attributes

Name	Type	Description	Required
name	string	Name of the frame	YES

Example

```
<decoderDef name=”myDecoder”>
      <frame name=”myFrame”/>
</decoderDef>
```

<outputDevice>

Parent

<decoderDef>
<decoderBCH>
<decoderVCDU>
<decoderSub>
<decoderHdr>
<decoderStr>
<decoderCmd>

Attributes

Name	Type	Description	Required
name	string	Name of the decoder	YES
frameID	integer	ID of the frames to send to this decoder	NO

Example

```
<decoderDef name=”myDecoder”>
      <outputDevice name=”myDevice” id=”1”/>
</decoderDef>
```

<outBuffer>

Parent

<decoderDef>
<decoderBCH>
<decoderVCDU>
<decoderSub>
<decoderHdr>
<decoderStr>
<decoderCmd>

Attributes

Name	Type	Description	Required
size	integer	Max size of any frame that could be sent to this decoder	YES

Example

```
<decoderDef name=”myDecoder”>
      <outBuffer size=”1024”/>
```

```
</decoderDef>
```

<query>

Description

Defines a single query to be sent whenever send_query "query name" is sent. For a more detailed description, please refer to Sending Queries.

Note: If a query's return value is a state, the associated item to hold the return value (i.e. the receiveDest) must have a state conversion associated with it.

Parent

<hydraDef>

Attributes

Name	Type	Description	Required
name	string	Name of query	YES
string	string	String query to be sent to hardware	YES
decoder	string	queryDecoder used to interface with the hardware	YES
receiveDest	string	Item to hold query response. Type of receiveDest item must match type of query response, as well as be correctly interpreted.	NO
destination	string	Hardware to query	YES
interpreted	boolean	0 if query returns a string; 1 if any other return type	YES

Example

```
<itemDef name="chan1_volt" type="myFloat"/>
```

```
<query name=”meas_volt_1” string=”MEAS:VOLT:DC? (@1)”
decoder=”testDecoder” receiveDest=”chan1_volt”
destination=”powerSupply” interpreted=”1”/>
```