Fotech Solutions (Canada) Ltd.

Helios Communications

Prepared by: Steven W. Klassen

For Helios version: 4.X

September 5, 2014

Table of Contents

Executive Summary

Objective

The Helios unit from Fotech Solutions is capable - right out of the box - of communicating with third party systems. This document will describe the necessary protocols and configurations for obtaining information from the Helios system as well as how to control the Helios unit externally. We will conclude by describing how you could use this information in order to setup an external watchdog system that would automatically keep the Helios unit running and report on any problems.

Overview

There are four separate ways for getting the Helios unit to push information into your system.

XML Protocol

The XML protocol is the easiest to understand and implement. However it is also the least efficient. If all you want to receive is event notifications and heartbeats, this isn't a bad choice.

Web Socket Protocol

New in V4.3

This is the newest protocol and intended to be the most powerful and flexible of them all. It is a two-way protocol allowing the recipient to send requests as to just what sort of information they want to receive. It is however a text based protocol with the objects sent encoded as JSON objects. This is well suited for communication with web browsers (and is why we chose this as it allowed us to eliminate our Java applet) but it can also be handled nicely in other languages. Later in this document we describe a C++ library that we have found useful.

Binary Push

Deprecated

This is the method currently used to push information from a Helios system to a Panoptes system. It is being deprecated in favour of the Web Socket protocol. We will not discuss it in this paper other than to say if you have a Panoptes system this is how you would have your Helios units communicate with it.

External Comms

This is similar to the Web Socket Protocol except that it sends messages in a binary format. At the time of this writing we have not yet determined if we are going to deprecate it in favour of the Web Socket Protocol or if we will update it to be a binary representation of that protocol.

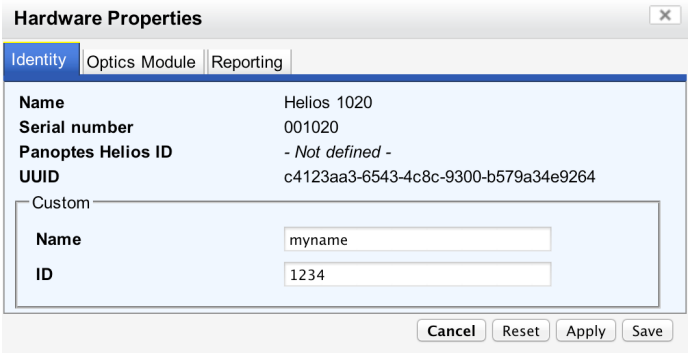
XML Protocol

Introduction

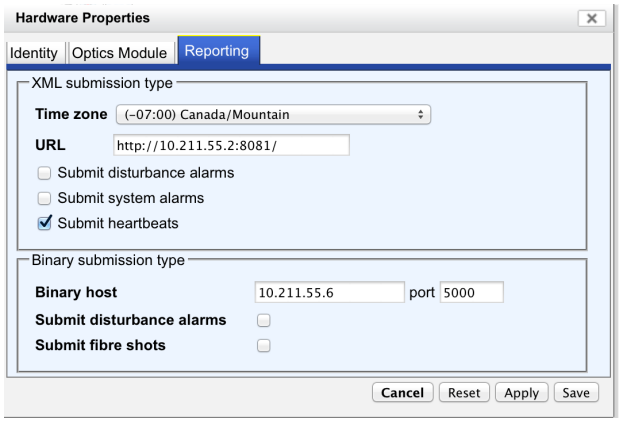
The XML protocol uses an HTTP connection to submit Events, Fibre Shots, and Heartbeats to a third-party server. This is the easiest of the protocols to implement as no binary manipulation is required and any number of off the shelf HTTP server packages can be used to implement the receiver.

In order to use this protocol you need to write an HTTP server that will accept an XML document in an HTTP POST. Then you use the HWI dialogs to configure the Helios unit to post the necessary data to your server.

Configuring in HWI



To configure the Helios unit for XML posting you click on the “Admin/Hardware properties...” menu item. The two tabs that you will need to use are “Identity” and “Reporting”. The “Identity” tab is used to identify the Helios unit to the outside world. The top portion contains information that is set at the factory and you cannot change. At the bottom you will find a section labeled “Custom” which is designed for you to use. In this section you can enter a name and an ID. Both of these are optional and are not used by any Fotech system. Instead they will be included in the XML messages posted to your server. In this way you can place your own custom identification into the messages. It is also OK to leave them out. That simply means no custom information will be included in the XML messages.



In the reporting tab you select what timezone the XML message timestamps should be displayed in, what URL they should be sent to, and you select which items you want to send. The URL should be a complete HTTP based URL. It may include a port number and a path so long as the server it is pointing to will accept that.

Changed in V4.5 Single URL and new time zone support.

Example XML Messages

Disturbance Alarms

Note: The disturbance alarms have not yet been converted to use the "Envelope" format or the time zone support.

Disturbance alarms - called "events" in earlier versions of the Helios software - consist of things that have been detected on the fibre. This includes walking and digging as well as fibre breaks. The following is an example of disturbance alarm sent from our system. (Note that it is still called an "event" in the XML.) The identification section identifies the Helios unit that detected the disturbance. The fibre-line-id is used to specify the fibre line ID from the Panoptes system that the event is tied to and will be empty unless the factory has configured your Helios unit to work in conjunction with a Panoptes unit. The "Custom" section is used to contain the name and/or ID specified by the user in the hardware properties dialog described above. (At some point the format of these messages will be changed to match the "Envelope" format used in the System Alarm and Heartbeat messages described later in this document.)

```
<?xml version="1.0" encoding="UTF-8"?>
<r:EventSubmit xmlns:r="http://www.fotechsolutions.com/schemas/repository-0.1.xsd" xmlns="http://
www.fotechsolutions.com/schemas/types-0.1.xsd">
    <r:Identification>
         <r:Source fibre-line-id="1" application="general"/>
         <r:Custom>
              <r:Name>custom name</r:Name>
              <r:ID>custom id</r:ID>
         </r:Custom>
    </r:Identification>
    <Event>
         <EventType name="fence" confidence="0.833333"/>
         <Time>2012-12-05T20:33:12.714121Z</Time>
         <EventTrackUUID>373ed1c2-cb7e-4edd-b19e-8cdc59588a7d</EventTrackUUID>
         <Magnitude>0.342677</Magnitude>
         <Width>5.064062</Width>
         <Velocity>0.000000</Velocity>
         <Acceleration>0.000000</Acceleration>
         <Location>
              <DistanceOnLine>346.92</DistanceOnLine>
         </Location>
    </Event>
</r:EventSubmit>
```

The other items in this message are as follows:

- EventType - specifies what type of event it is and how confident we are (between 0 and 1) that we have classified it correctly. As of the time of this writing the possible event types are "unknown", "fibre_break", "digging", "flat_wheel", "leak", "train", "walk", "gas_leak", "fence", "climbing", "fence_cutting", "rockfall", "vehicle", "heavy_equipment", "mech_digging", "generic", "cable", "theft", "lid_lift", "broken_rail". The confidence value is intended to be an indication of how confident we are that the event has been classified properly. It will be a real number between 0 and 1. However, as of the time of this writing its value is largely experimental and should not be relied on for any critical decision.

- Time - The time the event occurred. The format of this field is the standard XML xs:dateTime format. Although this format allows any time zone, our software always puts out the time in Zulu (hence the “Z” at the end).
- EventTrackUUID - If this element exists it identifies the track that the event belongs to. All events of the same track are assumed to have been caused by the same disturbance. For example, someone walking by the fibre would trigger multiple events on the same track. This is used by our user interface software (both Helios and Panoptes) to move the icon as new events in the same track arrive. It is also used by our Panoptes software to group multiple events into a single alarm.
- Magnitude - An indication of how large the event was in terms of its disturbance on the fibre. There are no units on this number and it is only useful in comparison with other events detected by the same Helios unit.
- Width - The size of the disturbance measured in metres.
- Velocity - An indication of how quickly the event is moving *along the fibre*. That last phrase is important as if the fibre is not straight (e.g. if it is wrapped around a pipe) the reported velocity will not equal the velocity along the ground. A positive value indicates movement away from the start of the fibre and a negative value indicates movement towards the start of the fibre. A zero (or nearly zero) value indicates a stationary event. The value is measured in metres per second.
- Acceleration - An indication of whether or not the event is getting faster or slower. The value is measured in metres per second squared.
- DistanceOnLine - The number of metres along the path of the fibre that the event occurred.

System Alarms

These are produced when there is a problem with the Helios system as a whole. At present there are four system alarms that may be sent - one when the disk drive is nearing capacity, one when the internal temperature gets too high, one if FDEL (the signal processing portion of the Helios unit) crashes, and one if we lose communications with the micro controller. All the system alarms will include the standard “Identification” section and a “Time” element showing the time of the alarm. The time will be in xs:dateTime format and converted to the time zone specified in the configuration.

Changed in V4.5
Timezone support

The disk drive and temperature alarms are sent when the available drive space, or the CPU core temperature, respectively passed a threshold. These alarms will be sent at regular intervals until the alarm condition itself has been resolved. (Note that these were called “alerts” in earlier versions of the Helios software. This name is still reflected in the XML messages.)

For temperature alarms the properties will list all the CPU cores whose temperature is too high. These will be of the form core_0_temp up to core_3_temp (for our existing hardware, newer hardware may have more cores).

For disk space alarms the properties will list the disks that are running out of space. These will be “system_disk” which refers to the root partition (“/” directory) and/or “data_disk” which refers to the data partition (“/HeliosData” directory).

```
<!-- sample temperature alarm -->
<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns="http://www.fotechsolutions.com/submit-envelope">
    <Identification>
        <Helios>
            <SerialNumber>001020</SerialNumber>
            <Name>Helios 1020</Name>
            <UUID>c4123aa3-6543-4c8c-9300-b579a34e9264</UUID>
            <ActiveChannel>1</ActiveChannel>
        </Helios>
        <Custom>
            <Name>myname</Name>
            <ID>1234</ID>
        </Custom>
    </Identification>
    <Alert>
        <Time>2012-05-29T21:58:09.709Z</Time>
        <Type>Temperature</Type>
        <Description>One or more CPU cores is reporting a high temperature. (Values in degC.)</
Description>
        <Property name='core_2_temp'>85.2</Property>
    </Alert>
</Envelope>

<!-- sample disk space alarm -->
<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns="http://www.fotechsolutions.com/submit-envelope">
    <Identification> ... same as for above example ... </Identification>
    <Alert>
        <Time>2012-05-29T21:58:09.709Z</Time>
        <Type>DiskSpace</Type>
        <Description>The system disk '/' or the logging disk '/HeliosData' is running
           low on disk space.</Description>
        <Property name='system_disk'>3.2</Property>
    </Alert>
</Envelope>
```

The third system alarm is sent if we lose communications with the micro controller. This alarm only applies to a two-box Helios system where the compute unit and the optics unit are two separate boxes. (The micro controller is a part of the optics unit.) The “MicroComms” alert will be generated either if the USB connection between the boxes is severed (in which case we get the alert very quickly as the compute unit receives a very specific error when it tries to access the optics) or if the optics box loses power (in which case it can take a minute or two before we get the alert as we have to rely on timeouts instead of a specific error message). In either case a system alarm is generated, a suitable line is added to the audit log, and the Helios system goes into an infinite loop attempting to reconnect to the micro controller.

The micro controller comms alert has no added properties.

This message is only sent once and is not repeated. You will know when the problem has been corrected when you start receiving heartbeats once again.

```
<!-- sample micro controller comms alarm -->
<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns='http://www.fotechsolutions.com/submit-envelope'>
  <Identification>
    <Helios>
      <SerialNumber>2002</SerialNumber>
      <Name>Helios 201</Name>
      <UUID>a51e0cfa-55c4-4359-961d-631dc3050d1e</UUID>
      <ActiveChannel>1</ActiveChannel>
    </Helios>
  </Identification>
  <Alert>
    <Time>2013-01-28T21:51:52.990Z</Time>
    <Type>MicroComms</Type>
    <Description>Lost communcation with the microcontroller, will attempt to reconnect.</
Description>
  </Alert>
</Envelope>
```

The FDEL crash alarm is sent if the FDEL process stops due to a system crash. It's properties include the process id (pid) of the process that crashed (useful if you are sending a problem report to Fotech as they can search for the pid in some of the logs) and the signal that was received (almost always 11).

Heartbeats

Heartbeats are small XML messages sent out periodically so long as the Helios unit is operating. Note that this differs from V4.4 and earlier where the heartbeats were only sent if FDEL was actually running. The heartbeat details will tell you if FDEL is running, if the laser is operating, as well as the available disk space and current CPU temperatures.

Changed in V4.5 New format and time zone support.

- Time - The time the heartbeat occurred. This will be reported in xs:dateTime format in the time zone specified in the reporting configuration.
- FDELStatus - Shows the current status of FDEL. This will be either "Running" or "Not Running".
- LaserStatus - Shows the current status of the laser. This will be "Laser On", "Laser Off", or "Locked Out, Laser Off".
- DriveUsage - Shows the percentage that the root drive and the data drive (typically /HeliosData) is used. This is a value between 0 and 100.
- Temperatures - Shows the current CPU temperatures.

```
<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns="http://www.fotechsolutions.com/submit-envelope">
    <Identification>
        ... same as described for System Alarms ...
    </Identification>
  <Heartbeat>
    <Time>2013-12-09T17:04:46.511-07:00</Time>
    <FDELStatus>Not Running</FDELStatus>
    <LaserStatus>Laser Off</LaserStatus>
    <DriveUsage>
      <RootDrive>24</RootDrive>
      <DataDrive>6</DataDrive>
    </DriveUsage>
    <Temperatures>
      <CPU core='0'>37</CPU>
      <CPU core='1'>35</CPU>
      <CPU core='2'>37</CPU>
      <CPU core='3'>34</CPU>
      <CPU core='4'>34</CPU>
    </Temperatures>
  </Heartbeat>
</Envelope>
```

Testing the XML Messages

Included with every Helios release is a Python script found in /usr/local/Fotech/bin/http_post_viewer that echoes any HTTP POST message that it receives. It provides a useful means of testing the XML posting of a Helios unit. When you run it you must specify the port it will listen to on the command line, for example, "`http_post_viewer 8081`". Then you enter the IP address and port number of the machine that you are running it on in the URL fields of the Reporting tab described earlier. (`http://10.255.254.165:8081/` in my test case). As alarms and heartbeats are generated you will see them echoed by this program. (This is also a good way for you to see exactly what it is that the Helios unit sends for a given message.)

Web Socket Protocol

Introduction

New in V4.3

We added this protocol in V4.3 in order to eliminate the need for the Java applets in our applications. Previously we used the “External Comms” protocol to talk to an invisible Java applet within our application which in turn would convert the binary messages into a JSON format that the web browser would then use. This had the advantage that we could send data via our most efficient protocol. But it had two drawbacks:

1. The Javascript code in our browsers cannot deal with binary data so the data had to be converted to a format that it could use, specifically JSON objects. So some of the advantage of the high efficient comms was lost by the need for this conversion.

2. But the bigger issue is that as Java applet support is continually shrinking in the industry, with Apple especially making it harder to deal with as each new OS update comes through.

It had become clear that we had to eliminate Java from our application.

Of course not everything in the system is a web browser and it is important that non-browser applications also be able to talk to our system. While they can continue to use the “External Comms” protocol (at least for the next few versions) this new protocol is also more powerful in that one in that the client can specify just what information they want to receive. (In contrast the “External Comms” protocol sends everything to everybody.) There are two C++ libraries that we use for this and can recommend. Both are freely accessible. For the Web Socket protocol itself the websocketpp library is available from http://www.zaphoyd.com/websocketpp and follows a BSD license. For producing and parsing JSON there is the rapidjson library available from http://code.google.com/p/rapidjson/ and follows an MIT license.

Overall Protocol

When the client first connects to the server (typically via port 57006 but that is configurable) it will immediately start receiving messages. The first message it will receive is a “Register” message from which it will need to extract the unique ID and keep track of it for future use. The next message it will receive is an “Identity” message which will identify the server that it is connecting to. From then on it will receive a variety of messages as the server decides to send them.

At any time the client can send a “Filter” message. This will be used to determine which messages the server will send. Note that the “Register”, “Identity”, “Heartbeat”, and “Shutdown” messages cannot be turned off. They will be sent regardless of the values in the “Filter” message. But no other messages will be sent until the “Filter” message is used to turn them on or off.

Note that all the times used in this protocol are specified in xs:dateTime format in the zulu time zone. Since this protocol is intended primarily for communication with other computers we have decided that it is more desirable to always have the times in the same format rather than supplying them in different time zones. As such we currently have no plans to convert these times to the time zone used in the XML message posts.

Specific Message Examples

The following sections describe each individual message. Note that the messages come through reasonably compressed with no spaces between items and all on a single (possibly very lone) line. In this document we show them in a somewhat more structured format to make them clearer. We also follow each line with a short description. These descriptions are *not* in the actual messages.

Note that every message contains the keys “object_type” and “object_type_version”. Also note that the ordering of the keys is not significant and can even change from one system or even from one message to another.

Alert

Called “System Alarms” in the user interface these are sent when there is something wrong. It will be one of the following four types.

Disk Drive Capacity

These are sent when either the system or data drive is nearing capacity. It will contain the following values.

- alert_type will be “DiskSpace”
- property_names will contain either or both of “system_disk” and “data_disk”
- property_values will contain the %full values (as strings) matches the order of the property_names

High Temperature

These are sent when one or more CPU cores are reporting a high temperature. High is defined as +5°C over the high temperature mark defined by the hardware. (You can read these by typing “sensors” on the Helios command line when logged in as the root user.) It will contain the following values.

- alert_type will be “Temperature”
- property_names will contain the core ids of the cores with high temperatures, e.g. “core_2_temp”
- property_values will contain the temperature in °C of the cores listed in property_names

FDEL Error or Crash

This is sent when FDEL crashes or otherwise stops with an error condition. It will contain the following values

- alert_type will be “FDEL Shutdown”
- property_names will be “reason”, “pid”, and “signal”
- property_values will contain a short description of why the exit, the process ID of the FDEL that stopped/crashed, and (if it was a crash) the signal that was received. These are typically of little use other than for reporting bugs to the Fotech support staff

Loss of Comms with the Micro-controller (Optics Unit)

This is sent when a two-box unit (where the optics unit is separate from the processing unit) losses communication between the two boxes. This may be due to the physical connection (USB cable) getting removed or severed, or due to a loss of power on the optics unit. The alert_type will be “MicroComms” and there are no additional properties.

```
{
    "object_type": "Alert",                  Type of message, will always be "Alert"
    "object_type_version": 3,                Version type, current version is 3
    "alert_type": "DiskSpace"                See the descriptions given above
    "time": "2013-03-21T20:45:32.739Z        Time it was detected (in xs:dateTime format)
    "num_properties": 2                      Number of properties in the message
    "property_names": ["system_disk", "data_disk"]
    "property_values": ["87", "25"]
}
```

Error

This message encompasses general notifications that are sent and displayed to the user. It includes not only errors, but also warnings and general information.

```
{
    "object_type": "Error",                  Type of message, will always be "Error"
    "object_type_version": 1,                Version type, current version is 1
    "time": "2013-03-21T22:11:05.452Z",      Time of the error (in xs:dateTime format)
    "source": "FDEL",                        Source of message (either "FDEL" or "CNCD"
    "code":4                                 Error code (see below)
}
```

The meaning of these messages depends on both the “source” and the “code”. Internally we interpret them as either general information messages (displayed briefly in the bottom status bar of HWI), warning messages (highlighted and displayed in the bottom status bar), or error messages (a dialog is popped up for the user to acknowledge, plus the message is highlighted and displayed in the bottom status bar).

The following Javascript code (which is used by the HWI system) describes the meaning of the various sources and codes and may be freely used and/or adapted for use in your own systems.

```
/*
# FILENAME:     helios_errors.js
# AUTHOR:       Steven Klassen
# DESCRIPTION:  Translates helios errors into readable messages.
# COPYRIGHT:    This file is Copyright (c) 2013 Fotech Solutions Ltd. All rights reserved.
*/


/*
 * Construct the errors object.
 */
var HeliosErrors = function() {
    this._levels = new Hash();
    this._levels['status'] = '';
    this._levels['warning'] = 'Warning: ';
    this._levels['error'] = 'Error: ';

    this._sources = new Hash();

    var fdel = new Hash();
    fdel.set(0, { level: 'status',  msg: null });
    fdel.set(1, { level: 'error',   msg: "Panoptes latency is too high" });
    fdel.set(2, { level: 'status',  msg: "Reporting queue size is too high" });
    fdel.set(3, { level: 'error',   msg: "Cannot write to log file" });
    fdel.set(4, { level: 'status',  msg: "Stopping" });
    fdel.set(5, { level: 'error',   msg: "CUDA execution failed" });
    fdel.set(6, { level: 'error',   msg: "FIR filter could not be found" });
    fdel.set(7, { level: 'warning', msg: "Missing or invalid property, recoverable" });
    fdel.set(8, { level: 'error',   msg: "Missing or invalid property, stopping" });
    fdel.set(9, { level: 'error',   msg: "Cannot allocate sufficient CUDA memory" });
    fdel.set(10, { level: 'error',  msg: "Acquisition card is missing or driver not installed correctly" });
    fdel.set(11, { level: 'error',  msg: "Cannot write to disk",
        advice: "Check that the drive containing /HeliosData is not full." });
    fdel.set(12, { level: 'error',  msg: "Error logging data",
        advice: "Check that the drive containing /HeliosData is not full." });
    fdel.set(13, { level: 'error',  msg: "Out of physical memory",
        advice: "You can try lowering the PRF or reducing the FFT size." });
    fdel.set(15, { level: 'warning', msg:"Calculating zero energy due to narrow analysis band.  No alarms will be generated",
        advice: "You can try widening your analysis band or increasing FFT size."});
    fdel.set(16, { level: 'warning', msg: "Attempting to log faster than the hardware will allow",
        advice: "You can try logging a shorter distance, logging at a slower rate (by decreasing PRF), or possibly upgrading to
faster hardware."});
    fdel.set(17, {level: 'error', msg: "Could not handle logging speeds, FDEL will shut down",
        advice: "You can try logging a shorter distance, logging at a slower rate (by decreasing PRF), or possibly upgrading to
faster hardware."});
    fdel.set(20, { level: 'error',  msg: "Acquisition driver is in an invalid state",
        advice: "Often this can be corrected just by restarting FDEL. If it keeps failing try powering the Helios unit off then on."
});
    fdel.set(21, { level: 'error',  msg: "Acquisition driver failed to start the data capture",
        advice: "You should try restarting the Helios unit." });
    fdel.set(22, { level: 'error',  msg: "Timed out while waiting to acquire data",
        advice: "Ensure that the trigger is correctly connected and that the trigger level is set correctly. Restarting the Helios
unit may help. If the problem persists you will need to contact Fotech support." });
    fdel.set(23, { level: 'error',  msg: "Processing could not keep up with the acquisition rate",
        advice: "You should try increasing the FFT size or the frame stacking factor." });
    fdel.set(24, { level: 'error',  msg: "Acquisition driver failed during a memory read",
        advice: "You should try restarting the Helios unit." });
    fdel.set(25, {level: 'warning', msg: "The acquisition system cannot keep up with the data production rate.",
        advice: "Acquisition has been restarted, however, some data will have been lost."});
    fdel.set(26, {level: 'warning', msg:"Processing cannot keep up with acquisition"})
    fdel.set(27, {level: 'error', msg:"A memory error has occurred when preparing for data acquisition. The allocated memory buffers
may be too large for the acquistion driver.",
     advice: "You should try reducing the FFT size, the fibre length, or the data Sample Rate."})
    fdel.set(28, {level: 'warning', msg:"Actual acquisition rate (PRF) differs from the requested rate by more than 10%"});
    fdel.set(99, { level: 'status', msg: "Starting.." });
    this._sources.set('FDEL', fdel);

    var cncd = new Hash();
    cncd.set(0, { level: 'status', msg: null });
    cncd.set(1, { level: 'error', msg: 'Could not open the playback file',
        advice: "Check that the correct drive has been mounted to /HeliosData and that the expected directory structure exists." });
    cncd.set(2, { level: 'status', msg: 'Found the Helios microcontroller' });
    cncd.set(3, { level: 'warning', msg: 'Still searching for the microcontroller' });
    cncd.set(4, { level: 'error', msg: 'Unimplemented command',
            advice: "This indicates a command that has not yet been implemented. This message should only be seen during product
development. NO CUSTOMER SHOULD EVER SEE THIS. If you are a customer and see this please contact Fotech support." });
    cncd.set(5, { level: 'error', msg: 'Failed to start/stop the laser',
            advice: "Either the laser is physically locked out or there is a problem with the hardware. Check the lockout key. If
it is not locked out you will need to contact Fotech support." });
```

```
    cncd.set(6, { level: 'error', msg: 'Failed to set the optics properties',
            advice: "One or more optics properties could not be set. This typically means there is a hardware problem. If this
continues to occur you will need to contact Fotech support." });
    cncd.set(7, { level: 'error', msg: 'FDEL returned error code',
            advice: "The FDEL process (the fibre monitoring process) has exited with an error condition. If this continues to occur
you will need to contact Fotech support." });
    cncd.set(8, { level: 'error', msg: 'FDEL has crashed',
            advice: "The FDEL process (the fibre monitoring process) has crashed. If this continues to occur you will need to
contact Fotech support." });
    cncd.set(9, { level: 'error', msg: 'Some properties are missing in the supplied file',
            advice: "The file you tried to open is missing some information that is critical for playback.  The file header will
need to be modified to add the missing properties." });
    cncd.set(10, { level: 'error', msg: 'License violation',
            advice: "It would appear you are trying to use functionality that you are not licensed to use." });
    cncd.set(11, { level: 'error', msg: 'Disk is too full',
            advice: "Either the system drive or the logging drive is too full and FDEL has been stopped as a precaution. Ensure you
have sufficient free space, then restart FDEL." });
    cncd.set(12, { level: 'error', msg: 'CPU temperature is too high',
            advice: "One or more CPUs is reporting a temperature that is over 10degC higher than its warning temperature. FDEL has
been stopped as a precaution. Allow the system to cool, then restart FDEL." });
    cncd.set(13, { level: 'error', msg: 'Lost comms with the optics unit',
            advice: "We have lost communication with the optics unit. Please wait while CNCD restarts in an attempt to restore
communication." });
    cncd.set(14, { level: 'warning', msg: 'Automatically restarting FDEL in 10 seconds...' });
    cncd.set(15, { level: 'warning', msg: 'Automatically restarting FDEL in 5 seconds...' });
    cncd.set(16, { level: 'warning', msg: 'Automatically starting FDEL in 30 seconds...' });
    cncd.set(17, { level: 'warning', msg: 'Automatically starting FDEL in 10 seconds...' });
    cncd.set(18, { level: 'warning', msg: 'Automatically starting FDEL in 5 seconds...' });
    cncd.set(19, { level: 'warning', msg: 'FDEL returned error code' });
    // messages 20-26 are monitored by the /views/channels/_optics_optimization_dialog.erb
    // to determine when to update the text of the dialog or when to close the dialog
    // if there are any more message relevant to the optics optimization process, please update that dialog
    cncd.set(20, { level: 'status', msg: 'Optimization has started, please wait...' });
    cncd.set(21, { level: 'status', msg: 'Optimization in progress, please wait...' });
    cncd.set(22, { level: 'warning', msg: 'Optimization was interrupted, no changes were made to the settings' });
    cncd.set(23, { level: 'error', msg: 'Optimization of the optics failed',
            advice: "The optimization failed to complete properly. This usually means there is not enough fibre (at least 1.5km is
required) or there is a physical problem with the optics unit. Check all connections and if they seem fine, contact Fotech support."
});
    cncd.set(24, { level: 'status', msg: 'Optimization has completed' });
    cncd.set(25, { level: 'status', msg: 'Performing sweep to find any TEC response, please wait...' });
    cncd.set(26, { level: 'error', msg: 'Optimization of the optics failed',
            advice: "The optimization has failed as we cannot compute the backscatter level. This typically means the fibre is too
short (at least 1.5km is required)." });
    cncd.set(27, { level: 'error', msg: 'The optics unit has been restarted',
            advice: "The optics unit has had a power cycle. Please wait while CNCD restarts in order to re-establish proper
communication." });
    this._sources.set('CNCD', cncd);
}

/**
 * Returns the full error hash for the given error JSON object.
 */
HeliosErrors.prototype.error = function(er) {
    var src = this._sources.get(er.source);
    if (!src)
        console.log("ERROR: Could not find a source for " + er.source + ".");
    else {
        var err = src.get(er.code);
        if (!err)
            console.log("ERROR: Could not find an entry for source=" + er.source + ", code=" + er.code + ".");
        else
            return err;
    }
    return null;
}

/**
 * Return the message that should be displayed for a given error JSON object.
 */
HeliosErrors.prototype.message = function(er) {
    var err = this.error(er);
    if (err)
        return this._levels[err['level']] + "(" + er.source + ") " + err['msg'] + ".";
    return null;
}

/**
 * Returns a long message including the short message plus some advice on what the
```

```
 * user can do as a response.
 */
HeliosErrors.prototype.longMessage = function(er) {
    var err = this.error(er);
    if (err) {
        var msg = this._levels[err['level']] + "(" + er.source + ") " + err['msg'] + ".";
        if (err['advice'])
            msg = msg + "\n\nSuggestion: " + err['advice'];
        return msg;
    }
    return null;
}



/**
 * Singleton instance of the errors object.
 */
var HELIOS_ERRORS = new HeliosErrors();
```

Event

These are called “Disturbance Alarms” in our user interface and are things that are detected along the fibre of our system. This can include fibre breaks, walking, digging, leaks, and so on.

Note that each event is largely independent of the others, with the exception of the event track. The event track is identified by a UUID (universally unique identifier) which has no inherent meaning other than that events with the same event track UUID are believed to belong to the same physical event (e.g. someone walking along the fibre).

As of the time of this writing the possible event types are "unknown", "fibre_break", "digging", "flat_wheel", "leak", "train", "walk", "gas_leak", "fence", "climbing", "fence_cutting", "rockfall", "vehicle", "heavy_equipment", "mech_digging", "generic", "cable", "theft", "lid_lift", "broken_rail".

What about the tags?

The tags section in this message will currently be empty when events are generated by the Helios unit. That may change soon and we will publish the changes as they are made. But the key purpose of the tags section is for third-party systems to add information to an event before passing it along. Panoptes, for example, is likely to add “latitude” and “longitude” tags after it has determined the position of the event on the map. This is also where custom code created for a specific installation may listen to the events on the Helios unit, add some tags, then pass the event on to some other system.

```
{
    "object_type": "Event",                 Type of message, will always be "Event"
    "object_type_version": 5,               Version type, current version is 5
    "channel": 1,                           Channel detected on, always 1 if no MUX available
    "event_type": "vehicle",                Type of event
    "confidence": 0.44329897,               How confident are we that the event type is correct
    "time": "2013-01-10T12:31:52.004Z",     Time of event (in xs:dateTime format)
    "event_track_uuid": "6fe3092c-2fcb-4d49-be5c-085d334e0992",
    "amplitude": 4.1086535,                 How large was the event
    "position": 672.8178,                   Position along the fibre, in metres
    "width": 1.362384,                      Width of the event, in metres
    "velocity": -0.040395174,               In metres/second (negative is moving towards start)
    "acceleration": 0                       In metres/second^2
    "time_series": {                        Time series object (optional)
        "sample_rate": 2000,                Sample rate of the time series
        "width": 1.2,                       Width (metres) of the data used to create the TS
        "data": [ 15.1, 1.6, ... 18.5 ]     The actual time series data
    }
    "tags": [                               This section allows for generic key-value tag values
        {                                   to be included in the message. Each 'tag' contains
            "key": "my-tag-name",           a key, a value, an optional units, and an optional
            "value": "the tag value",       visibility flag. The units and visibility flag are
            "units": "m",                   used by Panoptes when displaying the tag to the user.
            "visible": true                 If left out the units will be empty and the visiblity
        },                                  will be set to false. Note that visibility is a hint
        {                                   to Panoptes and has no meaning to the system as a
            "key": "an-invisible-tag",      whole.
            "value": "1234.5"
        }
    ]
}
```

Fibre Shot

Describes a single fibre shot. Presently only one "type" of fibre shot is sent to all users, so if user A changes the displayed fibre shot user B also sees it change, but in an upcoming release we will allow different clients to request different fibre shots.

```
{
    "object_type": "FibreShot",             Type of message, will always be "FibreShot"
    "object_type_version": 2,               Version type, current version is 2
    "display_type": "RAW_FIBRE_SHOT",       Type of the fibre shot display
    "time": "2013-01-10T12:31:25.201Z",     Time of the shot, in xs:dateTime format
    "channel": 1,                           Channel was taken on, always 1 if no MUX available
    "starting_position": 659.194,           Position on fibre, in metres
    "bin_size": 3.4059598,                  Size of each bin, in metres
    "num_amplitudes": 36731,                Number of values in the amplitudes array
    "amplitudes": [9352,11725,6126, ... ,8035,1518,3351]
}
```

Filter

The client sends this message to the server whenever it wants to change the messages that it receives. Note that the Register, Identity, and Heartbeat messages will be sent regardless of the filter settings.

The “connection id” is the unique ID that was given you in the Register message.

To specify what types of objects you want to receive, place their names in a list in the “object_type_filter” key. Alternately, you can specify a object type of “--all--” which will receive all events regardless of their type or to an empty array which will reset to the default settings. In this case the filter type line would look like `"object_type_filter": ["--all--"]`.

```
{
    "object_type": "Filter",                Type of message, will always be “Filter”
    "object_type_version": 1,               Version type, current version is 1
    "connection_id": 111,                   Your connection (unique) id.
    "object_type_filter": ["Event","Error","Status"]
}
```

Heartbeat

The heartbeat messages are sent regularly. They are used to provide some basic system status information as well as to ensure that the connection does not time out. These messages are always sent and cannot be turned off by the Filter message. Typically the heartbeats are sent out every 15 seconds (that is configurable) with additional Heartbeats being sent when there is a status change that would affect the information in a Heartbeat.

```
{
    "object_type": "Heartbeat",             Type of message, will always be “Heartbeat”
    "object_type_version": 4,               Version type, current version is 4
    "current_channel": 1,                   Current MUX channel, always 1 if no MUX available
    "fdel_status": 0,                       0=FDEL not running, 1=FDEL running, 255=unknown
    "laser_status": 0,                      0=laser off, 1=laser on, 2=locked out, 255=unknown
    "root_drive_used_percent": 23,          Percentage of disk space used on the root drive
    "data_drive_used_percent": 67,          Percentage of disk space used on the data drive
    "data_device": /dev/sda4,               Currently mounted data device
    "number_cores": 4,                      Number of CPU cores for which we measure temperature
    "core_temperatures": [65,66,64,68]      CPU core temperatures
}
```

Identity

The server will send this message shortly after the Register message. It will send it again anytime the identification information is changed. The “set at factory” items are set at the factory and cannot be changed by the users. There are two additional fields which can be set by the customer (user). For a description on how to set these see the section “Configuring in HWI” in the “XML Protocol” chapter.

```
{
    "object_type": "Identity",              Type of message, will always be "Identity"
    "object_type_version": 1,               Version type, current version is 1
    "name": "Helios 24",                    Name (set at factory)
    "serial_number": "00024",               Serial number (set at factory)
    "uuid": "45440f62-766c-46e3-97d1-f53f94823649,   Universally unique identifier (set at factory)
    "custom_name": "",                      Name (set by the customer)
    "custom_id": "12"                       ID (set by the customer)
}
```

Property

This message is sent when one or more property changes have been made in the database. The “connection_id” property is used to define which browser made the change and is the “unique_id” received in the “Register” message of the machine that made the change. This is important to ensure that you don’t react to changes that you yourself have made. A “connection_id” of 0 implies a change that was made by the server instead of by a user.

```
{
    "object_type": "Property",              Type of message, will always be "Property"
    "object_type_version": 1,               Version type, current version is 1
    "connection_id": 111                    Unique ID of the browser that made the change
}
```

Register

The register message is sent from the server whenever a client connects. It specifies an ID that the client will use to identify itself to the server when necessary.

```
{
    "object_type": "Register",              Type of message, will always be "Register"
    "object_type_version": 1,               Version type, current version is 1
    "unique_id": 111                        The unique ID (the "connection id")
}
```

Shutdown

The shutdown message is sent from the server when it is about to perform a clean shutdown. This may be used by the listeners to perform their own cleanup, should they so desire, without having to wait for the loss of the connection to be detected.

```
{
    "object_type": "Shutdown",                Type of message, will always be "Shutdown"
    "object_type_version": 1,                 Version type, current version is 1
    "reason": "system is shutting down"       This could be anything (or may not be included)
}
```

Status

The status messages are sent every few seconds while FDEL is actually running. They provide information on the current health and efficiency of the FDEL process. Note that not all of these are currently displayed to the user. Others are converted from their integer values into time values (by taking into account the current PRF) before being displayed to the user.

Warning: The "Status" message is primarily used for debugging and optimization purposes. Hence the details of what are sent in a given Status message change over time as different portions of our software submit different status messages. For this reason we do not actually document the Status message and we discourage its use in your software. If you wish to use it you will need to use a tool like wsterm to view examples of the Status message in order to determine what they mean. (They are fairly self-describing.) You will also need to write your code in such a manner that it will not fail if certain details are left out of a status message.

Testing the Protocol

Every Helios system comes with a program called /usr/local/Fotech/bin/wsterm. This stands for "Web Socket Terminal" and provides a command line terminal means of communicating with any web socket protocol. To use it type in "wsterm <url>" where <url> is the web socket protocol and address. Specifically to connect to the Web Socket Protocol on a Helios unit you would type in "wsterm ws://localhost:57006/".

The program will start echoing all the messages that are sent to it. Anything you type is then sent to the server.

External Comms Protocol

Introduction

The external comms protocol was added in order to overcome certain problems of the XML and binary push protocols. The advantages over XML should be obvious in that the binary nature of this protocol is much more efficient. In addition to the performance benefits, there is more information available using this protocol than that of the XML. While we will continue to support the XML protocol - primarily as it is useful for sending small amounts of information to remote locations - it will likely never grow to include all the information available in the binary protocols.

The main problem with the binary push protocol that the external comms was designed to overcome was that of allowing multiple recipients to connect without the need for a "middle man." For comparison, the binary push protocol was designed to push data to a Panoptes machine which in turn would serve that data to multiple sources. The external comms eliminates that second step allowing multiple connections to be made directly to the Helios unit.

Warning

Note this protocol has been superseded in our software with the Web Socket Protocol. At the time of this writing we have not decided if we will deprecate this protocol or if we will expand it to do everything that the Web Socket Protocol also does, but in a binary instead of text fashion. The advantage of this is that it would allow more efficient communication with non-browser applications. The cost is that we would have to maintain two protocols instead of one.

The Protocol

Overview

The protocol itself is quite simple. The Helios unit listens on port 57005. When it receives a connection it immediately sends out a 'Register' message. From that point on the Helios unit will write the various messages to that port. Hence when writing a receiver, you need to first make a connection, then listen and wait for messages.

If you receive no messages within a certain time frame you can consider this an indication that the system is not running and take whatever action you deem appropriate. In addition if you receive a message whose type appears to be garbage, you should take that as an indication that the comms have become garbled and disconnect, then reconnect to the port.

Message headers

All the messages used in this protocol contain a similar 16 byte header. When reading messages you should first read the header in order to determine the size and types of data to read for the remainder of the message.

Testing The External Comms

Each Helios units comes with a Python script called /usr/local/Fotech/bin/fmessage_viewer that echoes information about any message it receives. You can use this to view the messages posted by a Helios unit by pointing it at your Helios unit, for example, "`fmessage_viewer 192.168.20.60 57005`".

The Data Types

Note that all data is specified in network byte order. The Java language is already in network byte order so there are no special actions required, other than noting that since Java has no unsigned types you need to store the value in the next largest integer type (e.g. a uint08 needs to be stored in Java as a 16 bit integer). For C/C++ ntohl and its related functions can be used to ensure the proper ordering. In Python the unpack method is used using "!" as the first character. For other languages you will need to check your documentation to determine how to deal with network binary data.

The following are all the data types used in the messages of this protocol.

UINT08 - an unsigned integer of 1 byte. In C/C++ it is represented as a uint8_t and read simply by copying a single byte.

UINT16 - an unsigned integer of 2 bytes. In C/C++ this is represented as a uint16_t and converted to the local machine type using the ntohs function.

UINT32 - an unsigned integer of 4 bytes. In C/C++ this is represented as a uint32_t and converted to the local machine type using the ntohl function.

REAL32 - an IEEE single precision (4 bytes) floating point value in the byte order consistent with that of Java. In C/C++ this is represented by float and converted to the local machine by copying 4 bytes into a uint32_t, calling ntohl to convert it to the local machine ordering, then copying the resulting 4 bytes into a float.

REAL64 - an IEEE double precision (8 bytes) floating point value in the byte order consistent with that of Java. In C/C++ this is represented by a double. The functions ntohd and htond shown in this sidebar can be used to perform the necessary conversion.

TIME - A 17 character ASCII string containing the date and time in the form yyyymmddHHMMSSuuu all in zulu time. (uuu represents milliseconds)

UUID - A 36 character ASCII string containing a

```c
/* Determine the endianness of our architecture. */
const int _endian_i = 1;
#define is_bigendian() ( (*(char*)&_endian_i) == 0 )

/* Swap the bytes in a double if necessary. */
static double ntohd(double d) {
    double ret = d;
    if (!is_bigendian()) {
        uint32_t tmp1, tmp2;
        memcpy(&tmp1, &d, 4);
        memcpy(&tmp2, ((uint8_t*)&d)+4, 4);
        tmp1 = ntohl(tmp1);
        tmp2 = ntohl(tmp2);
        memcpy(&ret, &tmp2, 4);
        memcpy(((uint8_t*)&ret)+4, &tmp1, 4);
    }
    return ret;
}

static double htond(double d) {
    double ret = d;
    if (!is_bigendian()) {
        uint32_t tmp1, tmp2;
        memcpy(&tmp1, &d, 4);
        memcpy(&tmp2, ((uint8_t*)&d)+4, 4);
        tmp1 = htonl(tmp1);
        tmp2 = htonl(tmp2);
        memcpy(&ret, &tmp2, 4);
        memcpy(((uint8_t*)&ret)+4, &tmp1, 4);
    }
    return ret;
}
```

universally unique identifier. This will have been generated using uuid_generate followed by uuid_unparse_lower.

Message Types

This section describes all the message types that you may receive from a Helios unit. Note that even if you are not interested in certain types of messages, you will need to read them and throw them away as the current protocol sends all the messages to all the recipients.

Register

When you first connect to the port, this message will be sent to the browser.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 20.
10 bytes	ASCII	object type	'Register' - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Incremented each time we change the definition of an object. Currently 1.
4 bytes	UINT32	connection id	The unique id used to identify the browser connection.

Shutdown

This is not currently used in our protocol, but will probably be added soon. It is intended to flag that the server will soon shutdown and would allow clients to perform actions without having to wait to detect the loss of the connection.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 16.
10 bytes	ASCII	object type	Shutdown' - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Incremented each time we change the definition of an object. Currently 1.

Property

This is sent to all browsers when one or more property changes have been made in the database. The connection id field will be set to the connection id of the browser that submitted the changes. This will be zero if the changes did not come from a browser but from the system itself. (This typically occurs when the system is first started and the details of the hardware are obtained and stored.)

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 20.
10 bytes	ASCII	object type	'Property' - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Incremented each time we change the definition of an object. Currently 1.
4 bytes	UINT32	connection id	The unique id used to identify the browser connection that caused the change.

Heartbeat

This is used to provide an assurance to the user that something is happening (via a pulsing heart) as well as to make the communication robust. If the browser does not receive a heartbeat within a certain time frame it will close its connection and try to re-establish it. Unlike the XML version these heartbeats are sent so long as the system is alive whether FDEL is running or not.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 72+ncores for object type version=4. (71+ncores for object type version=3, 18 for object type version=2, 16 for object type version=1)
10 bytes	ASCII	object type	'Heartbeat' - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Currently 4.
1 byte	UINT08	FDEL running status.	0 = FDEL not running, 1 = FDEL running (*** only valid for version >= 2)
1 byte	UINT08	Laser status	0 = laser off 1 = laser on 2 = laser off, locked out (*** only valid for version >= 2)
1 byte	UINT08	Root drive space	A value between 0 and 100 showing the % used space on "/". (*** only valid for version >= 3)

Size	Type	Name	Description
1 byte	UINT08	Logging drive space	A value between 0 and 100 showing the % used space on “/HeliosData”. (*** only valid for version >= 3)
50 bytes	ASCII	Logging device	Describes the currently mounted logging device. (*** only valid for version >= 3)
1 byte	UINT08	Number CPU cores (ncores)	The number of CPU cores for which we will report temperatures. Will be 0 if we cannot determine temperatures on the hardware. (*** only valid for version >= 3).
ncores bytes	UINT08	Temperature for Core N	The temperature (deg C) of each core. (*** only valid for version >= 3).
1 byte	UINT08	Optical Channel	The current optical channel (always 1 for non-MUX systems). (*** only valid for version >= 4)

Identity

This message is sent out when a client first connects (right after the Register message) and again any time the Helios unit’s identify information is changed. These are found in the Hardware dialog of the UI as described in the XML Protocol chapter. *New in V4.3*

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 212.
10 bytes	ASCII	object type	Identity’ - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Currently 1.
50 bytes	ASCII	Name	Name of the Helios unit as defined by the Fotech factory.
30 bytes	ASCII	Serial number	Serial number of the Helios unit as defined by the Fotech factory.
36 bytes	UUID	Unique ID	Universally unique identify as defined by the Fotech factory.
50 bytes	ASCII	Custom name	Name as defined by the customer (may be empty).
30 bytes	ASCII	Custom ID	ID as defined by the customer (may be empty).

System Alarms

Previously called "alerts," these are submitted when something is wrong. For the possible values for alert type and the property names and values, see the description for the System Alerts in the XML protocol.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these for bytes. The value will be 52 + (n*65).. (24 for version=1, 25 for version=2).
10 bytes	ASCII	object type	'Alert' - padded to a length of 10 characters with ASCII character 0.
2 bytes	UINT16	object type version	Currently 3.
4 bytes	UINT32	alert_id	The id of the alert. (***only valid for version = 1 or 2)
4 bytes	UINT32	organization_id	The id of the organization who owns this alert. If 0 then it is a general alert that may be seen by all organizations. (***only valid for version = 1 or 2)
1 byte	UINT8	resolved_flag	If 0 then the alarm has not been resolved, 1 if it has been resolved. (***only valid for version = 2)
15 bytes	ASCII	alert type	The type of the alert. (***only valid for version >= 3)
17 bytes	TIME	time	The time of the alert. (***only valid for version >= 3)
4 bytes	UINT32	no. properties (n)	The number of properties for the alert. (***only valid for version >= 3)
n*15 bytes	ASCII	property names	The property names. There must be n of them, each 0 padded to 30 bytes. (***only valid for version >= 3)
n*50 bytes	ASCII	property values	The property values. There must be n of them, each 0 padded to 50 bytes. (***only valid for version >= 3)

Status

This data type specifies the current status of a running FDEL. (More types may be added in the future.) These are only generated while FDEL is running.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 93 (89 for version=1).
10 bytes	ASCII	object type	'Status' - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Incremented each time we change the definition of an object. Currently 2.
1 byte	UINT08	detection thread usage	A percentage integer ranging from 0 to 100.

Size	Type	Name	Description
1 byte	UINT08	reporting thread usage	A percentage integer ranging from 0 to 100.
1 byte	UINT08	identification thread usage	A percentage integer ranging from 0 to 100.
1 byte	UINT08	retrieval thread usage	A percentage integer ranging from 0 to 100.
1 byte	UINT08	feature extraction thread usage	A percentage integer ranging from 0 to 100.
4 bytes	UINT32	ShotSets Available	The Number of ShotSets still available in the buffer. (0 to 99999)
4 bytes	UINT32	shot sets awaiting detection	An integer ranging from 0 to 999999.
4 bytes	UINT32	reports waiting to be sent	An integer ranging from 0 to 999999.
4 bytes	UINT32	reports have been sent	An integer ranging from 0 to 999999999.
2 bytes	UINT16	shot sets held for copying	An integer ranging from 0 to 999.
2 bytes	UINT16	events awaiting identification	An integer ranging from 0 to 999.
2 bytes	UINT16	events are being processed	An integer ranging from 0 to 999.
4 bytes	UINT32	events have been processed	An integer ranging from 0 to 999999999.
1 byte	UINT08	network thread state	An integer ranging from 0 to 9.
1 byte	UINT08	logging thread usage	A percentage integer ranging from 0 to 100.
4 bytes	UINT32	logging thread write speed	An integer ranging from 0 to 999999.

Size	Type	Name	Description
2 bytes	UINT16	logging thread queue size	An integer ranging from 0 to 9999.
17 bytes	TIME	last FDEL startup time	The time FDEL was last started.
17 bytes	TIME	current time	The current time (generated by FDEL, not the time the message was received).
4 bytes	UINT32	backscatter level	The current backscatter level. (Only for version=2 or greater.)

Error

This data type is used to report an error. For a description of the meaning of the error codes, see the Error section in the Web Socket Protocol chapter earlier in this paper.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be 45.
10 bytes	ASCII	object type	'Error' - zero padded to a length of 10 characters.
2 bytes	UINT16	object type version	Incremented each time we change the definition of an object. Currently 1.
17 bytes	TIME	time of error	The time the error was detected (not the time the message was received).
10 bytes	ASCII	error source	Component that generated the error, zero padded to a length of 10 characters with the ASCII character 0. Can be either 'FDEL' or 'CNCD'.
2 bytes	UINT16	error code	This value will depend on the error source.

Disturbance Alarms

Previously called "events" these describe something that was detected due to a disturbance on the fibre.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these four bytes. The value will be computed as follows: 149 bytes for the event header (113 for version 4, 101 for version 3, 100 for version 2, 96 for version 1) add 16 + nts*4 bytes if a time series is included in the message add 16 + nfs*4 bytes if a fibre shot is included in the message add 4 + nt*310 bytes if event tags are included in the message
10 bytes	ASCII	object type	'Event' - padded to a length of 10 characters with ASCII character 0.
2 bytes	UINT16	object type version	Currently 5. Version 5 added the event track universally unique identifier. Version 4 added event width, velocity, and acceleration fields. Version 3 added the "has event tags" field as well as all the corresponding "et:" fields. Version 2 added the "event track id" field.
4 bytes	UINT32	id	Id of the event. Will always be 0 when coming from a Helios unit.
4 bytes	UINT32	fibre line id	The id of the fibre line the event occurred on.
4 bytes	UINT32	event track id	(Only if the version is 2 or greater.) Id of the event track. This will be 0 if no even track has been assigned.
36 bytes	UUID	event track uuid	(Only if the version is 5 or greater.) Event track UUID. Should be an empty string if no UUID is assigned.
10 bytes	ASCII	application	The name of the application type. (e.g. 'security')
15 bytes	ASCII	event type	The name of the event type. (e.g. 'unknown' or 'leak_started')
4 bytes	REAL32	confidence	How confident are we that the event type is correct. Between 0.0 and 1.0.
17 bytes	TIME	time	The time of the event.
4 bytes	REAL32	magnitude	The magnitude of the event.
4 bytes	REAL32	distance	The distance along the line in metres. This should be the distance of the centre of the event.
8 bytes	REAL64	latitude	Will be 0 for events coming from a Helios unit.

Size	Type	Name	Description
8 bytes	REAL64	longitude	Will be 0 for events coming from a Helios unit.
4 bytes	REAL32	event width	(Version 4 or greater.) The width of the event in metres.
4 bytes	REAL32	velocity	(Version 4 or greater.) The velocity of the event in m/s. A negative velocity indicates the event is moving towards the start of the fibre.
4 bytes	REAL32	acceleration	(Version 4 or greater.) The acceleration of the event in m/s
1 byte	UINT08	has time series	Flags to note if the event has a time series. If 0 there is none. If bit 1 is set (i.e. value == 1) a time series exists. If bit 2 is set (i.e. value == 2 or 3) a time series is included in the message. If a time series exists for the event the value should be set to 3 and the time series section included when CNCD posts this to the repository. The value will be set to 1 and the time series section will not be included when the event is passed on to the browser.
1 byte	UINT08	has fibre shot	Flags to note if the event has a fibre shot. If 0 there is none. If bit 1 is set (i.e. value == 1) a fibre shot exists. If bit 2 is set (i.e. value == 2 or 3) a fibre shot is included in the message. If a fibre shot exists for the event the value should be set to 3 and the fibre shot section included when CNCD posts this to the repository. The value will be set to 1 and the fibre shot section will not be included when the event is passed on to the browser. *Note that fibre shots included in events are being deprecated and will be removed in a future version. No new code will ever include a fibre shot.*
1 byte	UINT08	has event tags	(Only if version is 3 or greater.) Flag to note if the event has associated tags. If 1 then there are tags and the “et:” fields will be filled. If 0 there are no tags. Note that the push daemon itself may add tags so this value may change after receipt before the message is sent on to the browsers.
4 bytes	UINT32	ts: time series id	The id of the time series. Will be 0 for events coming from a Helios unit.
4 bytes	REAL32	ts: sample rate	The sample rate of the time series.
4 bytes	REAL32	ts: bin size	The bin size of the time series in metres.
4 bytes	UINT32	ts: no. samples (nts)	The number of samples in the time series.
nts*4 bytes	REAL32 array	ts: amplitudes	The time series samples.
4 bytes	UINT32	fs: fibre shot id	The id of the fibre shot. Will be 0 for events coming from a Helios unit.
4 bytes	REAL32	fs: starting position	The position of the first bin in metres.
4 bytes	REAL32	fs: bin size	The size of the bins in metres.

Size	Type	Name	Description
4 bytes	UINT32	fs: no. amplitudes (nfs)	The number of amplitudes in the fibre shot.
nfs*4 bytes	REAL32 array	fs: amplitudes	The fibre shot amplitudes.
4 bytes	UINT32	et: no. tags (nt)	The number of event tags. The remaining et: tags will be repeated nt times, once for each event tag. In these tags the index i will go from 0 .. nt.
4 bytes	UINT32	et: event tag id[i]	The event tag id. Will be 0 for events coming from a Helios unit.
40 bytes	ASCII	et: key[i]	The event tag key.
255 bytes	ASCII	et: value[i]	The value of the tag.
10 bytes	ASCII	et: units[i]	The units (if applicable) of the tag.
1 byte	UINT08	et: visible[i]	Set to 1 if the tag should be visible in the displays and 0 if it is used only internally.

Fibre Shots

This data type will describe a single fibre shot. These are the lines that make up a "sound field" display.

Size	Type	Name	Description
4 bytes	UINT32	message length	The total length of the message, including these for bytes. The value will be 107 + (n * 4). (57 + (n*4) for object type version = 1) The recipient should ignore any fibre shot received whose message does not include the number of bytes specified here (i.e. an incomplete message).
10 bytes	ASCII	object type	'FibreShot' - padded to a length of 10 characters with ASCII character 0.
2 bytes	UINT16	object type version	Incremented each time we change the definition of an object. Currently 2. (2 added the fibre shot type)
4 bytes	UINT32	id	Id of the fibre shot. Will be 0 for all fibre shots coming from a Helios unit.
4 bytes	UINT32	fibre line id	Id of the fibre line associated with the fibre shot.
4 bytes	UINT32	event id	Id of the event. Will be 0 for all fibre shots coming from a Helios unit.
4 bytes	REAL32	starting position	The position of the first bin of the fibre shot in metres.
4 bytes	REAL32	bin size	The size of the bin in metres.
17 bytes	TIME	time	The time the fibre shot was taken.

Size	Type	Name	Description
4 bytes	UINT32	no. amplitudes (n)	The number of amplitudes (bins) being reported in this fibre shot.
n*4 bytes		amplitudes	An array of n REAL32 values containing the amplitudes of the fibre shot.
50 bytes	ASCII	fibre shot display type	This is the type of fibre shot (typically the display type). In FDEL terms this is the value of “soundField.display.type” at the time the fibre shot was created.

Raw Streaming Protocol

Introduction

New in V4.5.3

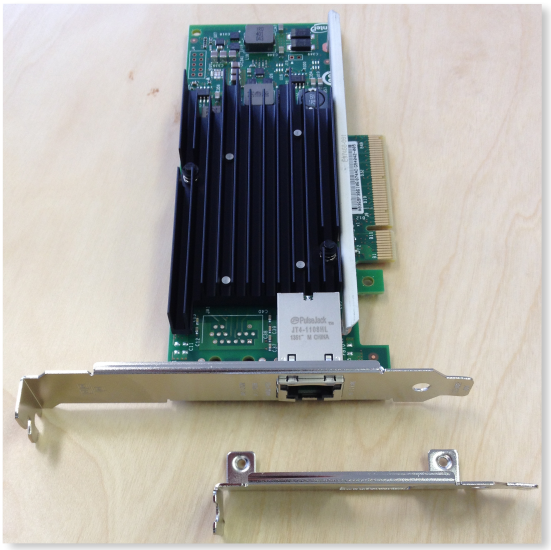
For most of our customers the Helios unit hides the bulk of the data. The raw data comes from the optics unit (more specifically from the A/D card in the processing module of the unit) at a rate of almost 1TB/hour. This is a data rate that most would find daunting. For this reason the Helios unit is designed to rapidly analyze this data, make some decisions based on that analysis, and then throw it away. The exception to this has always been our raw logging of the data which stores it to disk. And, yes, to do that you need either an SSD or a RAID system in order to perform sustained writes at those speeds.

In keeping with that data rate, the communications protocols described so far in this document all occur at a lower rate. They are sending out the results of the analysis instead of the full raw data. However as of V4.5.3 you now have the option to obtain the raw data itself and process it however you desire.

***Warning**: The protocol described below allows the raw data to be streamed to any single TCP/IP port. However a gigabit network is too slow to maintain this data rate. In order to use this functionality you will need to have installed our 10GigE networking option. Without that high speed network, the protocol will work but it will end up dropping most of the raw data and you will only see a small fraction of it appear on your machine.*

Installing the High Speed Network Card

***Note**: If you purchased a Helios unit with the high speed streaming option it will already contain the necessary card. But you should have also received a second card to be installed in your recipient machine. If you did not receive such a card (because your system already contained a 10GigE card) then you can ignore this step.*



The card we have chosen to supply is a single-port 10GigE card made by Intel. It is called the "Intel(r) Ethernet Converged Network Adapter X540-T1."

Step 1 - Select the bracket. You will notice that it ships with two brackets, one made to fit full size 4U boxes and one made to fit smaller boxes. If your machine is a smaller box the first thing you

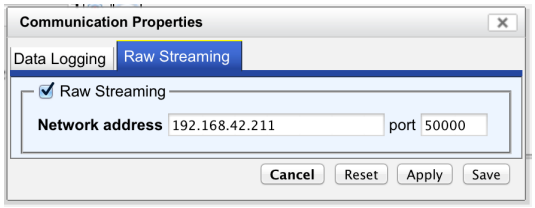
will need to do is to remove the large bracket and install the small bracket.

Step 2 - Install the card. Since we don't know what your box looks like, we can't really describe this other than to say that you need to power off your box, open it up, and install the card into a spare PCI Express slot.

Step 3 - Install the drivers. This will of course differ based on your OS. For some OSes (recent versions of Windows, for example) you may not have to install anything. For others (most versions of Linux, for example) you will almost certainly need to install a driver. The drivers should have been provided to you on a USB stick together with the card. If that USB stick was not included, you can download the latest drivers from http://www.intel.com/support/go/network/adapter/cd.htm. Once you have the drivers downloaded, open the readme.txt file and follow the instructions for your OS.

Configuring the Raw Data Streaming

Once the hardware has been installed, you can turn on the raw data streaming by clicking on the "Admin/Communications…" menu item then on the "Raw Streaming" tab within that dialog.



Fill in the network IP address (or a name that the Helios unit knows how to resolve) and the port that it should connect to and either "Apply" or "Save" the changes. The next time that the processing is started (i.e. the next time you press the "Start Helios" button) it will attempt to stream the raw data to that port.

The Protocol

The overall protocol is a TCP/IP based socket. The remote system (your system) needs to listen to the port specified in the properties dialog described above. When the Helios unit starts processing it will attempt to connect to that port. If it succeeds it will start streaming the messages described below. If it does not succeed it will quietly ignore the request and start processing, but it will keep trying. When the port becomes available the streaming will start. When the port ceases to be available, the system will continue processing but without the streaming. If the port becomes available again the streaming will once again proceed. In other words, the Helios unit is very resilient to the existence of or absence of the port on your system.

In addition, the Helios unit is resilient to the speed of the processing on your unit. If you cannot read the data as fast as we write it, the Helios unit will cache a small amount of the data but will start dropping data rather than ceasing to stream the data. You will be able to determine when this is happening by examining the header values of the data packets (the details are described below.

There are presently three messages that are produced by our system. These are called 'Header', 'Data', and 'Terminator'. The 'Header' package is sent when the Helios unit first connects to the port. Then 'Data' packages are send repeatedly until either the Helios processing is stopped or we lose the connection to the port. Finally a 'Terminator' package may be sent when the Helios unit stops processing. After receiving a 'Terminator' package you should not see any more packages arrive until you see a new 'Header' package.

The Messages

The communication that you can expect to receive on this port will consist of distinct message packages. Each message starts with a short header containing the package type and a timestamp describing when it was sent. Following that header comes the message contents itself, which naturally is distinct for each message package type.

The common header consists of two lines of ASCII text containing the following:

```
{package type}\n
{timestamp}\n
```

The {package type} describes the type of the package and will be one of 'Header', 'Data', or 'Terminator' (without the quotes).

The {timestamp} will be the time the package was sent formatted in an xs:dateTime format. For example, '2014-08-28T08:31:12Z'.

Following the header will be the payload of the message which will be one of the following.

Header

This message package will be the first one sent when a new connection is made. It will also be the first message sent after a 'Terminator' message. (i.e. You won't ever receive 'Terminator' followed by 'Data', if you receive 'Terminator' there will probably be a delay but when processing begins again there will be a new 'Header' sent.

The header message contents consists of a set of ASCII key value pairs of the following form. Note that the "=" may or may not have spaces around it - you should make no assumption either way but should separate the line on the "=" and then trim the resulting key and value before using them.

```
{key}={value}\n
```

For example,

```
acquisition.laserPulseRate=2000\n
```

The first line of this header package will have a key of "length" and the value will indicate the length of the remainder of the message in bytes. You can use this to determine how many bytes you need to read in order to obtain the remainder of the message package.

The following example shows what a complete Header package looks like. One warning though, while you can assume that these keys will all be there you should not assume they come in any particular order other than the guarantee that the 'length' key will be the first one.

```
Header\n
2014-09-01T15:54:26.621647Z\n
length = 465\n
acquisition.numSamplesPerShot=147\n
acquisition.numShotsPerShotSet=512\n
acquisition.alazar.sampleRate=150MSPS\n
acquisition.alazar.channelA.trigger.delay=0\n
acquisition.fibreLightSpeed=202000000.000000\n
acquisition.fibreRefractiveIndex=1.484121\n
acquisition.fibreBoxDelaySec=0.000000e+00\n
acquisition.laserPulseRate=20000\n
user.monitor.startDistance=0\n
user.monitor.endDistance=40000.0\n
channel.zeroPoint_m=0.000000\n
channel.name=Optical Channel 1\n
channel.length_m=40000.000000\n
```

Data

After the 'Header' messages you will receive numerous 'Data' messages. The 'Data' messages contain the raw fibre shot data as read from the ADC card.

Warning: *At the time of this writing these Data messages will consist of the FFT size number of raw fibre shots. But you should make no assumptions along those lines as when we add the ability for overlapping FFTs, this block size will unlikely remain tied to the FFT size. You should rely strictly on what the 'Data' message tells you and not on any pattern you think you see.*

The first part of the 'Data' message will consist of an ASCII header similar in form ({key}={value} pairs) to the 'Header' message. Here is an example of what one would look like:

```
Data\n
2014-09-01T15:54:26.621821Z\n
lengthHeader = 341\n
Counter = 0\n
StartShot = 0\n
EndShot = 512\n
StartBin = 0\n
EndBin = 147\n
RelativeShot = 0\n
sessionID = 0\n
timeReference.time = 2009-04-17T15:15:00Z\n
timeReference.counter = 0\n
timeReference.intervaluS = 50\n
NumberOfShots = 512\n
NumberOfSamples = 147\n
SamplingFrequency_measured = 20000\n
SampleSize_m = 0.6733329892\n
TimeOfFirstSample = 2009-04-17T15:15:00Z\n
lengthData = 150528\n
```

You should not make any assumptions regarding the ordering of these values other than we guarantee that the 'lengthHeader' key will be the first one in the list and the 'lengthData' key will be the last one in the list. The 'lengthHeader' key will give the number of bytes that must be read (after that line) in order to read the remainder of the header. And the 'lengthData' is the number of bytes that must be read (after that line) in order to read the entire raw data section (which

follows the header section). (It will be equal to NumberOfShots*NumberOfSamples*2 but we explicitly include it so that you don't have to make that computation.) While most of these key/value pairs should be straightforward, a few of them need further comment.

Counter - This is an integer which will be incremented with every Data package that is sent. If the numbers you are receiving are not consecutive it is an indication that some of the data is being dropped. This is typically caused by one of the following problems:

1. The network connection is too slow. This level of communications requires a 10Gig-E connection. If you are running on a slower connection data is almost certainly going to be dropped.

2. The client is not reading the data fast enough. If your processing does not keep up with the rate the data is arriving (i.e. if your software is not reading as fast as we are writing) then data will be dropped. You will need to optimize your code - probably adding CUDA or other high performance processing to your machine.

3. The Helios unit is not sending the data fast enough. In theory this should not be happening. If you think this is the case you can reduce the data by either shortening your monitor start and end or by lowering your PRF. In either case if this becomes necessary you should contact Fotech Support as our systems are supposed to be able to write at these speeds.

sessionID - When streaming live raw data (which would be the norm) this will always be 0. When streaming previously recorded data (which would be primarily for testing purposes) every non-sequential movement in time will increment this value. If the playback is left running without interruptions this would indicate the number of times it has been replayed. But it will also be incremented if the user skips ahead or backwards in the time of the playback.

timeReference - This group of key values is used to pin a shot counter to a (presumably accurate) time stamp together with a best known actual PRF that can be used to get a timestamp of any shot in the temporal vicinity. Specifically the formula *timeReference.time + (startShot - timeReference.counter) * timeReference.intervaluS* will give you the time of the first shot set in the array to our best known accuracy. It is done this way rather than just giving you the time of the first shot set explicitly so that you know which portion of the timestamp is accurate (the timeReference.time) and which is an estimate (the timeReference.intervaluS). This estimate is based initially of the requested PRF but changes over time as the actual PRF becomes known more accurately. Note that TimeOfFirstSample will be close to this value, but will be less accurate as it is recorded only to the second, not to the uS.

The final section of the 'Data' package is the raw fibre shot data itself. It consists of (NumberOfShots * NumberOfSamples) values of 16 bit unsigned integers in Intel byte ordering. *Note that this differs from other binary communications in this document which are all in network byte ordering. Due to the high speed nature of the raw data streaming we decided to break from this standard and keep the raw data values in Intel byte ordering. If your receiving machine does not support this ordering, you will need to byte swap appropriately.* The values will come in "row major" order meaning that the first (EndBin-StartBin) values will be the first fibre shot, followed by the second, and so on until (EndShot-StartShot) fibre shots have been read. (You can also read them in a block based on the 'lengthData' value and separate them out after.)

Remember that these packages will come rapidly. For example, if you have a PRF of 10000 you will get a total of 10000 fibre shots per second. These will be grouped into sets of fibre shots (i.e. you don't get one 'Data' package for each shot). Currently this grouping is based on the FFT size meaning that for an FFT size of 256 you would receive 10000/256 (about 39) 'Data' packages a second. (Once again, don't assume that the block size will be the FFT size. Just assume that they will come rapidly and that they will be relatively large.)

Due to the speed these messages come, it is recommended that you have a thread that receives them and buffers them somewhere and another thread that actually processes them. There is buffering that occurs on the sending side, but it is designed to be "lossy" and will drop 'Data' packages if the system determines that they are not being read as quickly as we are trying to write them. You can detect when this is occurring by examining the StartShot and EndShot values of the 'Data' package. In particular if the StartShot of package N is greater than the EndShot of package N-1, then some data has been dropped.

***Note**: For reasons we will not go into at this time, we decided that EndShot and EndBin would not be inclusive. That is, EndShot is one past the last shot in the data and EndBin is one past the last bin in the data. Hence you should see that the following relationships always hold…*

NumberOfSamples = EndBin - StartBin
NumberOfShots = EndShot - StartShot, and
*lengthOfData = NumberOfShots * NumberOfSamples * 2 (since it is 16-bit binary unsigned integer data)*

This also implies that if no data is being dropped, the StartShot for package N should equal the EndShot of package N-1.

Terminator

The terminator message has no additional contents beyond the header itself. Its purpose is to signal that a processing run has completed. Note that you should not depend on receiving this package. If the connection is closed before processing terminates, then you will not receive this package. Your code should be robust enough to handle any of the following cases:

1. A Terminator package is received. Indicates that processing is done and you should handle it appropriately.
2. The connection is closed. You should treat this the same as a Terminator package. This isn't necessarily an error condition - it just means that the connection was closed before the Terminator got sent.
3. A new Header package is received. This shouldn't occur and indicates that somewhere a package got missed. But your code should be robust enough to handle it anyway. Treat it as though you had just received a Terminator followed by a Header.

Example Program

The USB stick that you should have received with the drivers will also contain a small C example program that you can use when writing your own client for this protocol. You will find it in the file Docs/streamviewer.c. Note that it has been written assuming C99 in a POSIX environment. It is intended to be example code only, but if you choose to use it directly on a Windows machine you will need to determine the Windows equivalent of some of the POSIX system calls.

On a Linux machine compile the program using “gcc streamviewer.c -o streamviewer”. On a Mac use “clang streamviewer.c -o streamviewer”.

If you are on a windows machine you will need the assistance of Windows developers to get the example program to compile and run. (One way to do this is using the “cygwin” package to make the Windows machine look more like a POSIX compliant machine.)

Control Protocol

Introduction

While most of the control of a Helios unit is performed via the Helios Web Interface (HWI) there are a number of items that third-party software may wish to control. This can be done by sending commands to the Helios unit on port 36832. The commands are sent as small XML documents written directly to a socket (i.e. this is just XML, not XML+HTML). When sending a command you should 1) open a connection to the port, 2) send your message, and 3) close the port. You should not keep the port open or send multiple commands.

If the message is parsed successfully, then “OK” will be sent followed by a newline. If there is a problem such as in improperly formatted message or an unsupported command then “ERR: “ followed by a brief error message, followed by a newline will be returned.

Note that the “OK” that is returned specifies only that the message was received an understood. It does not guarantee that the message will succeed in being carried out. For example, suppose that the laser is physically locked out. Then if you send the “start_laser” command you will get the “OK” message but the command will fail. Failures will be broadcast as CNCD error codes to those listening on the external comms interface. Successes will trigger a heartbeat which will send the appropriate change in state across the external comms interface.

Message Format

The messages you send are ASCII text messages containing a header, an XML wrapper, and an XML command. The commands are described in the next section, but the overall message will look like the following:

```
Protocol: 1.1
Message Length: xx
<message>...the command goes here...</message>
```

The protocol must always be 1.1 and the message length should contain the length (in bytes) of everything from <message> to </message> including those tags and the trailing newline (which is also required). The Helios unit will first read the header, then will read lines until it has read at least the number of bytes specified, then will attempt to parse the message as an XML document. If it gets that far, then it will process the command.

Supported Commands

FDEL and laser control

```
<start_laser/>
```

This is used to start the laser. Note that starting FDEL will automatically start the laser. However stopping FDEL does not automatically stop the laser. If the command fails CNCD error 5 (could not start/stop laser) will be sent, otherwise a heartbeat will be triggered.

```
<stop_laser/>
```

This is used to stop the laser. If it fails CNCD error 5 (could not start/stop laser) will be sent, otherwise a heartbeat will be triggered.

```
<start_fdel/>
```

If the laser is not already running, this will start the laser. Then it will start FDEL monitoring using the current properties as most recently submitted via the Helios Web Interface. After starting a heartbeat will be triggered. If FDEL fails to start then either a CNCD error 7 (FDEL exiting with an error code) or an FDEL error will be sent.

```
<stop_fdel/>
```

This will stop FDEL processing, assuming it is processing. This will always trigger a heartbeat and should never result in an error code. (i.e. We can always stop a process, we cannot always start one.)

Dynamic Properties

```
<dynamic_properties>
  <property key="...valid FDEL property name...">...value...</property>
  <property key="...valid FDEL property name...">...value...</property>
  ...repeated for each property...
</dynamic_properties>
```

This is the most complex of the supported commands. It allows some properties to be changed dynamically while FDEL is running. Note that this requires a fairly intimate knowledge of FDEL properties and how they are related to the stream and the zones. The following properties are the simplest ones to change in this fashion.

Simple dynamic properties

soundField.display.rate (specifies in Hz approximately how fast the sound field data is sent)

logging.rawShotSets.enabled (should be TRUE or FALSE to turn logging on or off)

logging.rawShotSets.range_m (should be the logging range in metres, e.g. 100:200 will log from 100m to 200m)

logging.rawShotSets.filename (should be the fully qualified path name of the file to log to. Typically this starts with /HeliosData but it can be any path that is mounted and read.)

(Note that the *logging* properties are typically all set in a single message - or at least they should all be specified when you turn *enabled* to TRUE.)

If you want to control other properties dynamically you should contact Fotech support for a more in depth discussion as doing this incorrectly can give you results that you may not expect.

Hint: Any property that can be set dynamically through the user interface can be set dynamically through this API. But the relationship between the property names and the zones and streams is non-trivial and goes beyond the description in this document. But if you need this functionality we will be happy to assist your technical staff in getting it set up.

Forcing System Checks

The temperature and disk space status checks take place at regular intervals. You can request these checks at specific times via this protocol. If any problem is found during the check an appropriate system alarm will be generated and sent out via all registered protocols.

```
<check_disk_status/>
```

This will force an immediate check of the disk space status.

```
<check_temperature/>
```

This will force an immediate check of the CPU temperatures.

Optimizing the optics

This command will cause the Helios to stop monitoring and enter into an optics optimization routine. This routine will modify the TEC temperature searching for an optimal value. During this process, which takes a few minutes, a number of error messages will be submitted in order to provide progress information to any listeners. *New in V4.2.0*

```
<start_optimization/>
```

This will cause the optimization process to begin.

```
<stop_optimization/>
```

If an optimization process is currently running, this will force it to stop and reset the optics to the values that were current before the optimization process began.

Example: Writing a Watchdog

Introduction

While we put great effort into making the Helios system robust and reliable, the reality is that things can go wrong. For many applications it is not sufficient to simply know that the system has gone down, you would like to do something about it, possibly even having another process keep track of the system and automatically restart the components when necessary. This is often called a watchdog process. If your Helios unit was purchased as part of a system including a Panoptes unit, the Panoptes unit will almost certainly have had a watchdog process configured to monitor the health of the various components - including the Helios units. In this chapter we show how you can use the protocols described above to write such a watchdog for monitoring a Helios unit.

For this example we will assume one Helios Unit and one other machine which will run the watchdog system. We will discuss the comms and control items required, then we will finish by writing a working watchdog code using Python.

Very important disclaimer! The code produced here is an example program that illustrates the techniques involved. It is not - I repeat **not** - a production quality watchdog to be used in a live system. The difference between a sample code such as this one and a production quality code is many thousands of hours of programming required to deal with all the exceptional conditions that may occur in a live system.

Internal Watchdog

The Helios unit has a small watchdog component built into it which will monitor the system health and will interrupt processing if either of the following occurs:

1. If either the logging drive (/HeliosData) or the root partition (/) start to get too full the watchdog will stop FDEL. This is to ensure that the system and the data do not get corrupted by completely filling the system.

2. If any CPU gets significantly hotter than it's warning temperature the watchdog will stop FDEL. (The FDEL processing on the CUDA cards is the most significant source of heat.)

If you would prefer to monitor these yourselves it is possible - but not recommended - to disable the internal watchdog. You will need to contact Fotech Support in order to accomplish this.

The remainder of the internal watchdog is tasked with seeing what is running and reporting it to the outside world as described earlier in this paper.

The Pieces

There are a number of pieces that need to come together in order to produce our watchdog.

First, we need a configuration. For this example the configuration will consist of hard-coded variables in a well defined location in the Python program. For a real system some form of configuration file would be a better solution.

Second, we need to know when the system is operating. This could be done using either the XML or the External Comms protocols. For our example we will use the External Comms protocol as it is quite easy to deal with in Python. Essentially we will create a thread that will connect to the external comms port and wait for heartbeats. If we receive heartbeats that say FDEL is not running we know the system is healthy but not being monitored. If we do not receive heartbeats within an expected time then we assume that there is something wrong and the system needs to be restarted.

Third, we need to know how to control the system. There are three controls we will refer to. The first is the XML control protocol described earlier in this paper. The second control will be to setup SSH to allow a "soft" restart of the system. The third control will be to use iAMT to force a "hard" restart of the system. (We will describe the “hard” restart but the sample watchdog code does not implement it.)

The XML Controls

The key XML control that you will need is the <start_fdel/> command. Our watchdog will send this command whenever it detects that the Helios system is healthy (i.e. we can talk to it) but FDEL is not running.

Restarting the Box Remotely

Of course starting FDEL is not enough. We also want to handle the cases where the Helios unit itself gets “stuck” and needs to be restarted. There are two ways that we can do this.

Soft Restart Using SSH

Obviously it would be a significant security risk if just anyone was allowed to remotely restart a machine. Fortunately the ssh protocol allows us to use digital signatures on the specific machines in order to limit this. (The following description is based on a Unix like system such as Linux or Mac OS. On a windows machine you will need to install an appropriate SSH package in order to accomplish this task. Your IT department should be able to assist you with that task.) A more detailed description of the technique described here can be found at http://www.eng.cam.ac.uk/help/jpmg/ssh/authorized_keys_howto.html.

Step 1. We will start opening up two shell windows, one on the Helios unit and the other on the watchdog machine. On the Helios machine you need to be the root user. On the watchdog machine you need to be the user who will be running the watchdog. Typically this would be the root user, but for this example it is just the normal user of my machine. For this example the Helios unit has an IP address of 192.168.42.101. The IP address of the watchdog machine is not important.

Step 2. Our goal is to be able to have the watchdog machine run a remote command (the restart) as the root user on the Helios machine. So let's start with trying to get a simple directly listing (the 'ls' command). On the watchdog machine type in

```
ssh root@192.168.42.101 "ls"
```

The first time you do this you will probably be told that the authenticity can't be established and asked if you want to continue. Type in "yes". You will only need to do this once. After you have confirmed the authenticity by typing "yes" the watchdog machine will be able to make ssh connections to the Helios unit until the Helios unit hardware is changed at which time you will be given a suitable message and have to reconfirm your answer. This is done to ensure that an attacker cannot place his machine between yours and pretend to be the Helios unit (called a man in the middle attack).

After you type "yes" you will probably be asked for a password. This is a problem. The watchdog needs to run without user intervention so you can't have someone ready to type in passwords, but neither do we want to store the root password to the Helios units on the watchdog machine as that would be a significant security risk. Fortunately the ssh protocol provides us a solution using digital certificates. At this point just hit Cntl-C to abort and go back to the command line prompt.

Step 3. The next step is to setup our digital certificates which will allow the watchdog to make an ssh connection to the Helios unit without having to prompt for a password. On the watchdog machine we create a digital certificate by doing the following.

```
cd ~/.ssh
ssh-keygen -f watchdog -C 'Watchdog key' -N '' -q
scp watchdog.pub root@192.168.42.101:.ssh/watchdog.pub
```

This will create two files. 'watchdog' is your private key which you shouldn't share with anyone and 'watchdog.pub' is your public key which we will place on the Helios unit to allow the access we require. You will also have to enter the root password of the Helios unit for the scp command (which is a secure copy of the file onto the Helios unit).

Step 4. Next we need to tell the Helios unit to allow your new key to be authorized. So on the shell window of the Helios unit do the following.

```
cd ~/.ssh
ls
```

You should see two files, "known_hosts" which you can ignore and "watchdog.pub" which is the file you copied in step 3. You may also see an "authorized_keys" file. If you do then you will need to edit it and append the contents of "watchdog.pub". If you do not then you can simply type the following command.

```
cp watchdog.pub authorized_keys
```

Step 5. Finally let's test our result. Back on the watchdog machine type in the following command.

```
ssh -i ~/.ssh/watchdog root@192.168.42.101 "ls"
```

Now the system should not prompt you for a password but instead see a listing of some files on the Helios unit.

If successful our setup is now complete and the watchdog process will be able to use ssh to send the reboot command to the Helios unit at the right time.

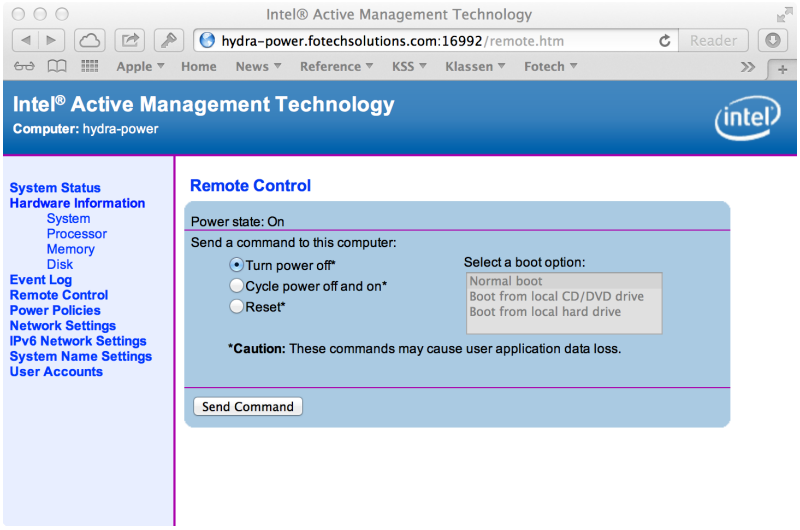
Hard Restart Using iAMT

Newer "two box" Helios systems include motherboards that have iAMT enabled. This is a mechanism provided by Intel that allows you to obtain information about a system as well as to power it on or off remotely. The iAMT details will have been configured at the factory and the relevant details included on your system fact sheet.

In this example we will use iAMT to power the system off and then on again if it has not responded to the soft restart mentioned above. Note that you would not want to use the hard restart initially as powering the system off while it is actively performing its duties can corrupt the system drive. So this should be the last resort if the system does not respond to the soft restart.

(*What if I have an older non-iAMT machine?* Older hardware may include a separate "Black Box" remote power switch as an extra option. These do not provide the running information that iAMT provides, but they can be used to cycle the power of a Helios unit. Configuring and using a "Black Box" is not a part of this paper. If you have purchased this option it will have been configured for you by the Fotech staff.)

The main tool that a user would use to control iAMT is simply a web browser. Point your web browser to the IP address that was setup for you (this should be described in the documentation that came with the Helios unit), supply the required username and password and you will see an extensive web based system monitoring user interface. (e.g. http://192.168.20.21:16992) The key panel is the “Remote Control” panel which allows you to cycle the power on the system - even if the main system is “down”.



Of course to use this in a watchdog we need the ability to control it via some API or command line procedure. We can do this using a tool called “amttool”. Panoptes machines will have this installed by default (since they use it in their watchdog process) but Helios units do not. You can download and install this tool on any Linux machine by using the command (as root) “yum install amtterm” (amttool is a subset of what gets installed). For instructions on other architectures see the Intel iAMT instructions. (Searching for iAMT on Wikipedia gives the most up to date instructions.) The amttool program (it’s actually implemented as a Perl script) allows us to send the commands powerup, powerdown, and powercycle to the iAMT hardware on the Helios unit. (e.g. “env AMT_PASSWORD=somegivenpassword amttool 192.168.20.21 info” will produce some info on the current status of the Helios unit)

Writing the Watchdog

Part 1: Configuration

Our configuration is quite simple in this example. There are really only four things that we need to configure - the IP address of the Helios unit we are interested in, the two ports on that machine that we need to connect to, and the number of seconds we expect there to be between the heartbeats. For the external comms protocol the number of seconds between heartbeats is typically set to 5 seconds. For our watchdog program we will store these two values in variables near the start of the program.

```
helios_ip = '192.168.42.101'
pushdaemon_port = 57005
control_port = 36832
expected_time_between_heartbeats = 5
```

Part 2: Controlling the Helios Unit

For this example program there are two helios control commands that we require - one to restart fdel, and one to reboot the box. These are both found in the helios_control class.

Restarting FDEL is done by sending a message to the control port of the Helios unit. We don't actually publish the full API for the messages on this port as they are subject to change from one version of the software to the next. But the "<start_fdel>" message that is used here in the "restart_fdel" function will remain for all of the 3.4 versions.

Rebooting the helios unit itself is a little more interesting. We can't just send a command to the control port because the process that monitors that port may be one of the things that has died. So instead we use the ssh protocol as described above in order to run the "reboot" command as the root user. The "restart_helios" function shows how this is done. Note that we pause after performing the restart. The reason for this is that the restart is not instant and it is possible to reconnect to the system after sending the reboot command but before the system has actually gone down. This can lead to confusion between the systems, so we pause to ensure that we do not attempt to reconnect until we are reasonably certain the machine has shut down.

Part 3: Monitoring Heartbeats

The "heartbeat_monitor_thread" isn't as complex as it may seem at first, and I have thoroughly commented it in the code, so I won't expand on it here in great detail. Essentially what it does is connect to a port on the Helios machine and wait for messages. It should receive a Heartbeat every 15 seconds and will receive other messages as well. We look for the heartbeats and if they tell us that FDEL is not running then we attempt an FDEL restart.

If we receive no communications for a period of time, then we assume that something has gone significantly wrong with the machine and we perform the reboot procedure.

Part 4: Turning this into a Production Quality Code

In order to move this from an example program to something that is production quality you would need to do a number of things.

First, you would need better configuration than just variables. Second, you would probably want it to support multiple helios units, each being observed in a separate thread, and each being able to restart without affecting the others. Third, instead of simply restarting things you would probably want to log the information somewhere (this program just writes it to the standard output device) and possibly even send notifications to someone. Fourth, you would need to be more exact in the exception handing rather than making the broad assumptions that this code does. Fifth, you may want to add pauses to see if a situation will correct itself, and you may want to limit the number of times that a restart would be allowed. In particular if the system gets into a truly corrupted state one could get an infinite loop of FDEL restart attempts when instead you probably want to reboot the box. Similarly you would not want an infinite loop of reboot attempts, but after a number would likely want to give up, disable the machine, and notify something that there is a serious problem.

I mention these just to emphasize that this is an example program. If you use it in a production environment it will almost certainly fail at some point.

Part 5: Moving it to Windows

This was developed on a Mac system which is a reasonably complete Unix platform. As such it includes tools - such as python and ssh - that don't come by default on a Windows system. If you are developing your watchdog on a Windows system you may want to consider a different set of tools and use this program only as an example.

```
#!/usr/bin/python

#  sample_watchdog.py
#  Helios
#
#  Created by Steven W. Klassen on 2012-06-04.
#  Copyright (c) 2012 Fotech Solutions (Canada) Ltd. All rights reserved.
#
# This file contains a sample watchdog process written in Python. It is intended as an
# example of how to write such a process and comes with no warranty or support.

import commands
import signal
import string
import threading
import time
from socket import *
from struct import *


##
## Configuration. In a real system the configuration should likely be read from
## a file or perhaps from a database of some sort. But for this example we simply
## hardcode the items we need.
##

helios_ip = '192.168.42.101'            # The IP address of the helios unit
pushdaemon_port = 57005                 # The port we connect to for the external comms protocol
control_port = 36832                    # The port used to send control commands to the Helios
expected_time_between_heartbeats = 5    # The number of seconds we expect between heartbeats

##
## Helios control object. This object will handle the restart controls necessary
## when we determine that something is wrong.
##

class helios_control:
    def __init__(self, heliosIP, controlPort):
        self.ip = heliosIP
        self.controlPort = controlPort

    # Attempt to restart FDEL by sending the start message to the control port. Note that
    # the control port comms are not described in this document and are largely considered
    # internal and subject to change, but you can use this message fairly safely.
    def restart_fdel(self):
        print "CONTROL: restart fdel!"
        msg = """Protocol: 1.1
Message Length: 33
<message><start_fdel/></message>
"""
        sock = None
        try:
            sock = socket(AF_INET, SOCK_STREAM)
            sock.connect((self.ip, self.controlPort))
            sock.send(msg)
        except Exception as ex:
            print "!!! Error trying to start FDEL: ", ex
        finally:
            if sock:
                sock.close()
```

```
    # Attempt to restart the helios unit using ssh to send the restart command. Note that
    # this requires that ssh be configured to allow a remote computer to do this as described
    # in the Helios Communications manual.
    def restart_helios(self):
        try:
            print "CONTROL: restart helios!"
            cmd = "ssh -i ~/.ssh/watchdog root@%s \"reboot\"" % self.ip
            commands.getstatusoutput(cmd)
            time.sleep(20)      # Don't even try to reconnect for a little bit.
        except Exception as ex:
            print "!!! Exception: ", ex
            print "!!! Error attempting to restart Helios."


##
## Heartbeat monitoring thread. This thread will attempt to connect to the external
## comms port of the helios unit and will wait for heartbeats. Each time we receive
## a heartbeat we check it to see if FDEL is running. If not then we attempt to restart
## FDEL. If we receive no communications at all for 3 times the expected interval
## between heartbeats, then we assume the unit has a more significant problem and
## we attempt to reboot it.
##
class heartbeat_monitor_thread(threading.Thread):
    def __init__(self, heliosIP, heliosPort, controlPort, expectedTimeBetweenHeartbeats):
        self.ip = heliosIP
        self.port = heliosPort
        self.sock = None
        self.control = helios_control(heliosIP, controlPort)
        setdefaulttimeout(expectedTimeBetweenHeartbeats * 3)
        threading.Thread.__init__(self)

    def stop(self):
        self._Thread__stop()

    def run(self):
        while True:
            # If we are not connected to the Helios unit, we will continue try to connect
            # in 5 second intervals until we are successful.
            while not self.sock:
                try:
                    self.sock = socket(AF_INET, SOCK_STREAM)
                    self.sock.connect((self.ip, self.port))
                    print "Connected to the helios unit %s" % self.ip
                except Exception as ex:
                    self.sock = None
                    print "!!! Exception: ", ex
                    print "!!! Could not connect to the helios unit %s, will retry in 5 seconds..."
% self.ip
                    time.sleep(5)

            # Read a message from the socket.
            try:
                (msgLength, objType, objVersion) = unpack('!I10sH', self.sock.recv(16))

                print "Received %s, version %d, length %d." % (objType, objVersion, msgLength)

                # If we receive a heartbeat telling us that FDEL is not running then we
                # restart FDEL. We may also want to close the socket (reopening it the next time
                # through our loop) so that we don't get multiple heartbeats queuing up if the
                # FDEL restart takes more than a heartbeat interval. But that is not handled
                # by this sample program. This would be important if you used a short
```

```
                # heartbeat interval (say less than 5 seconds).
                #
                # Another possible problem to handle is the case where FDEL starts right away
                # but then quickly dies (this can happen if the Alazar driver leaks memory).
                # One way to handle this would be to keep a counter of the number of restart
                # attempts since the last successful one, performing a restart_helios() if
                # there are too many FDEL restarts. (This entire function of automatically
                # restarting FDEL is a useful enough concept that we may consider building
                # it into CNCD. But for now it needs to be done via an external watchdog
                # process such as this one.)
                if objType.startswith('Heartbeat') and objVersion >= 2:
                    (fdelStatus, laserStatus) = unpack('!BB', self.sock.recv(2))
                    if fdelStatus == 0:
                        print "-- FDEL is not running will attempt to restart it"
                        self.control.restart_fdel()

                else:
                    # Throw away the rest of the message.
                    remaining = msgLength - 16
                    while remaining > 0:
                        data = self.sock.recv(remaining)
                        remaining = remaining - len(data)

            except:
                # If we receive an exception we assume it means that we have lost comms with
                # the helios unit and need to do a full restart on it. A proper watchdog should
                # likely be more specific in just what exception triggers this (i.e. don't
                # assume that all exceptions were comms errors) and you may want to pause
                # for a bit and see if it comes back on its own just in case it was a temporary
                # network glitch. Those cases are beyond this example program.
                print "-- Lost comms with %s" % (self.ip)
                self.sock = None
                self.control.restart_helios()


##
## Setup a signal handler so that we can use Cntr-C to stop the threads.
##
thr = None
def signal_handler(signum, frame):
    print 'Caught: %d' % signum
    thr.stop()

signal.signal(signal.SIGINT, signal_handler)


##
## And start the watchdog.
##

print 'Starting the watchdog on %s' % helios_ip
thr = heartbeat_monitor_thread(helios_ip, pushdaemon_port, control_port,
expected_time_between_heartbeats)
thr.start()

while thr.is_alive():       # Standard python hack to allow the signals to be caught. thr.join()
    thr.join(5000)          # without the timeout ignores the signal handling so the Cntr-C would
thr.join()                  # be ignored.

print 'Exiting the watchdog.'
```