 **Stockell™ Healthcare Systems, Inc.**

**Interface Training Manual**

**Purpose**

The purpose of this document is to equip members of the Stockell Support team with the knowledge and resources to troubleshoot interface issues in an effective and timely manner. This document will contain the following:

* [Explain what interfaces are.](#what_are_interfaces)
* [Describe the MonitorApp](#monitor_app)
* [Briefly explain what each interface’s function is](#interface_function)
* [Describe the interface configuration files](#config_files)
* [Describe strat files, msgrules, and segdefs.](#strats_msgrule_segdefs)
* [Describe the interface record routing](#record_routing)
* [Viewing an HL7 message](#viewing_message)

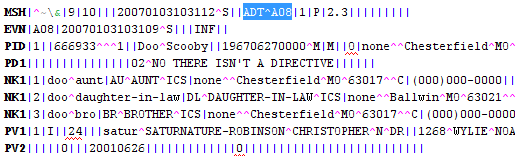
**What are interfaces?**

Interfaces are processes that run behind the scenes which have two main purposes: 1) Send messages out of InsightCS to other systems 2) Receive messages from other systems into InsightCS. We will refer to these as Outbound and Inbound interfaces, respectively.

These ‘messages’ are generally created as a result of some sort of update made within the system. For example, in InsightCS, an outbound message will be created as a result of any of the following actions: a new registration entered in, a transfer to a different room/bed, discharging a visit, any sort of change made to a visit (change in admit date, patient type change, insurance update), along with a vast number of other actions.

While not controlled by InsightCS, messages are generally sent inbound into InsightCS from similar updates made in other systems.

These ‘messages’ that are sent/received are referred to as *HL7 messages*. An HL7 message (see below for example) is a standardized messaging system that is used throughout the healthcare industry to communicate between different healthcare systems. Every HL7 message contains a message type, which identifies the functionality of the message. For example, the message type of the HL7 message below is *ADT^A08*, which signifies an update to a visit. InsightCS supports many other message types as well. Please see the [Interfaces Technical Design Doc](http://srvmoss/sites/intranet/Docs/Technical%20Design/Interfaces/Interfaces.doc) for details.



Another characteristic of an HL7 Message is the *segment name*. This is generally a three character identifier at the beginning of each line within a message. For example, the segments in the above message are *MSH*, *EVN, PID*, *PD1*, *NK1*, etc… . The segment name serves as a grouping of related information. For example, the *PID* segment contains Patient Identification information; the *NK1* segment refers to the *Next of Kin* information (or Relatives).

In each segment, there are individual elements, which are separated by the ‘|’ character. Each element represents a specific piece of information. For example, the *PID.18* field (which is the 18th element in the *PID* segment) refers to the Visit Number; *PID.5* refers to the patient name. Please refer to the [Interfaces Technical Design Doc](http://srvmoss/sites/intranet/Docs/Technical%20Design/Interfaces/Interfaces.doc) for a complete listing of what each segment/element points to.

Each element can also contain sub-elements, which are separated by the ‘^’ character. An example of this is an address field, in which the specific items within an address (street, city, state, zip) would be separated by the ‘^’ character.

Both the inbound and outbound interface have specific rules that are used, which tells the system how to create an HL7 message (outbound), and how to process a message (inbound) and store the information to the database. These rules are defined in strategy files, which will be addressed later in this document.

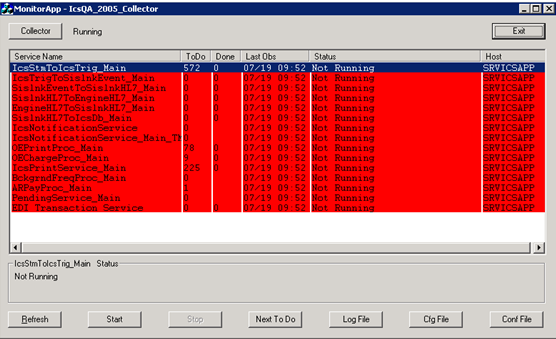
**What is MonitorApp?**

MonitorApp is an application that is used to monitor the current status of the Interfaces and Background Services. It displays the following pieces of information:

* The name of the service/interface
* The ‘ToDo’ count (or records that have yet to be processed)
* The ‘Done count (or the number or records that we processed over the last minute.
* ‘Last Obs’. This is the last time that the ‘ToDo’ and ‘Done’ counts were updated.
* ‘Status’ – represents the current state of the service/interface (running, stopped, starting, etc…)
* ‘Host’ – the control workstation that the service/interface is running from.

There are several buttons on MonitorApp, which will perform specific actions when clicked:

* Refresh – this will refresh the Status column (the ToDo, Done, and Last Obs columns are refreshed automatically every minute.)
* Start – Starts the interface
* Stop – Stops the interface
* Next To Do – This button currently is only functional for the SislnkEventToSislnkHL7, SislnkHL7ToEngineHL7, and SislnkHL7ToIcsDb\_Main interfaces. It is used to skip the current record that is being processed, and move on to the next record. This button should be used when the interface is in a ‘Not Running’ status.
* Log File – Displays the log file. You have the option to display the ‘tail’ (very end of the log), or the entire log. (If viewing the tail, you can refresh it by doing a *Ctrl+R*)
* Cfg File – Display the .cfg file (details will be provided later in this doc)
* Conf File – Display the .conf file (details will be provided later in this doc)



Figure

The Collector – The Collector is a service (normally named CollectorService) whose purpose is to keep MonitorApp functioning. This service is responsible for displaying all of the information in MonitorApp. If the Collector is not running, MonitorApp will not be functional, as the information will not be updated.

Generally speaking, the Collector has no effect on any of the interfaces, as they can run even if the Collector is not running. That being said, there is one case where the interfaces are tied to the Collector. One of the functions that the Collector performs is to check and see if the interfaces are up and running. If it determines that an interface has stopped (for example, due to an error), it will attempt to restart the interface automatically. This is the one way that the Collector directly impacts each interface.

At the top of MonitorApp, you will find a Collector button. By clicking the button, you will have the options to start/stop the Collector, and view the Collector log and .cfg files.

**COLLECTOR database – what is it used for?**

The Collector database’s sole purpose is to store all of the information that is used by MonitorApp. The Collector database primarily uses two tables: the SERVICE\_MSTR table, and the REGISTRY table.

* **SERVICE\_MSTR**: This table contains a listing of each service to be displayed on the MonitorApp.
* **REGISTRY**: This is the table that stores all of the necessary information that is used by MonitorApp. In viewing this table, you will see data primarily grouped by the following fields: SUBSYSTEM\_ID and KEY\_ID. The SUBSYSTEM\_ID is the service name. For each SUBSYSTEM\_ID, you will see many records with a different KEY\_ID. Each KEY\_ID value represents a specific function. For example, the KEY\_ID of CfgName represents the ‘Cfg File Name’. MonitorApp will look at the VALUE column to determine what Cfg File it needs to display when the ‘Cfg File’ button is clicked.   
  Not all KEY\_ID’s are used by every interface. There are some KEY\_ID values that are used only by a select number of interfaces. That being said, many KEY\_ID values are shared by all of the interfaces. They include the following:
  + **CfgName**: This tells MonitorApp what Cfg file to display when the ‘Cfg File’ button is clicked.
  + **CfgPath**: This tells MonitorApp the path of the Cfg file. This is necessary so it knows where to pull the Cfg File from.
  + **DbType**: This is the database type that the service gets its data from (Sislnk/InsightCS)
  + **HostName**: This is the machine name that MonitorApp will engage with. For example, when you click the Stop or Start button on MonitorApp, it will stop/start the service on that machine.
  + **KeepRunning**: If VALUE is set to TRUE, MonitorApp (through the Collector Service) will start the service back up automatically if it stops unnaturally (something other than the user clicking the ‘Stop’ button)
  + **LogName:** This tells MonitorApp what Log file to display when the ‘Log File’ button is clicked.
  + **LogPath:** This tells MonitorApp that path of the Log file. This is necessary so it knows where to pull the Log File from.
  + **NextRecord:** This tells MonitorApp whether or not a record can be skipped (Valid or Invalid). At this time, the only interfaces with a value of ‘Valid’ should be the Sislnk interfaces.
  + **PollRate:** This tells the Collector how ofter (in seconds) to update the ToDo/Done columns.
  + **Type:** This signifies the type of the interface. These values are fixed, and should never be changed.

**What does each interface do?**

There are 6 main interfaces – 4 are responsible for sending messages outbound, and 2 are responsible for receiving/processing messages inbound. The IcsStmToIcsTrig, IcsTrigToSislnkEvent, SislnkEventToSislnkHL7, and SislnkHL7ToEngineHL7 interfaces are outbound. The EngineHL7ToSislnkHL7 and SislnkHL7ToIcsDb interfaces are inbound.

**Outbound**

**IcsStmToIcsTrig –** When activities are performed within InsightCS, depending on the specific function performed, a Sistalk message will be created, which is stored in the INF table. (These functions include add/updating a visit, performing a transfer, discharge, discharge cancel, admission cancel, etc…). The specific message will look similar to figure 2 below. This interface reads the message, does some processing, and eventually creates an XML Trigger message, which is stored in the INF\_TRIGGER table (figure 3).



Figure 2



Figure 3

**IcsTrigToSislnkEvent –** This interface takes the XML Trigger message (generated from IcsStmToIcsTrig) and converts it to a XML Event message. (Figure 4). It reads the message from the INF\_TRIGGER table, and creates the XML Event message, writing it to the TRANSACTIONS table in the Sislnk database.

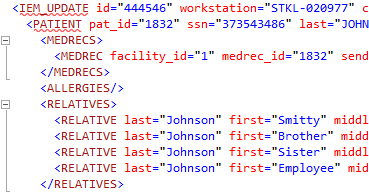


Figure 4

**SislnkEventToSislnkHL7 –** This interface takes the Event XML message (from the TRANSACTIONS table in the Sislnk database) and converts it to an HL7 message. The HL7 message is written to the TRANSACTIONS table as well.

**SislnkHL7ToEngineHL7 –** This interface takes the HL7 message and sends it outbound to the interface engine or to another healthcare system.

**Inbound**

**EngineHL7ToSislnkHL7 –** This interface receives HL7 messages from the engine (or other healthcare system) and writes them to the TRANSACTIONS table in the Sislnk database.

**SislnkHL7ToIcsDb –** This interface takes the HL7 messages that were stored from the EngineHL7ToSislnkHL7 interface and processes them, updating the InsightCS database based on the contents of the message.

**Configuration files…**

Each interface relies on configuration files, which tell the interface how it needs to operate. As mentioned above, there are two types of configuration files – cfg and conf.

**Cfg files –** Cfg files define specific settings for each interface. They have three main functions:

1. Specify the log file path (highlighted in red in figure 5 below)
2. Specify the maximum size of the log file in kb (highlighted in blue below). When writing to the log file, if the size starts going over this limit, it will simply truncate information at the top of the log to keep the size under the defined limit.
3. Specify the types of messages written to the log file (highlighted in green). The specific log message types are as follows:
   1. BROADCAST – The most basic messages, such as if the interface started/stopped.
   2. FATAL – Will be written if an interface stops for any reason.
   3. ERROR – Will be written if the interface cannot write certain information to the database, due to invalid data, incorrect configuration, etc… . ERROR messages will not stop the interface; it will continue to process.
   4. INFO – Informative messages as to what the interface is doing at a given time.
   5. TRACE – Similar to INFO messages, but a lot more specific.

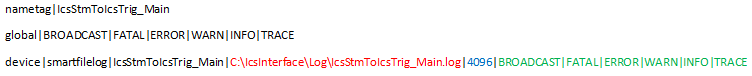
**

Figure 5

NOTE: All interfaces have an associated cfg file.

**Conf files –** Conf files tell the interface specifically how to operate. Figure 6 below shows an example of a conf file. The interface will read this file from bottom-up, and process each line, one at a time. Once it is completed with the bottom-most line, it will pass it off to the next line above it, and continue to do that until it reaches the top line, in which it will then roll back down to the bottom.

Each line contains several values. The first value is the specific .cpp file that it uses (*tm\_SislnkDbRAdaptAscii* in the last line). The second value is the solution that the .cpp file is located in (*SvsSislnkAdapter – see figure 7 below*). The remaining values are the parameters that the .cpp file is using. Those specific parameters are identified in the .cpp file, in the *Init* routine (see figure 8 below).

When processing the .cpp file, it will execute the *Receive* routine. Within that routine, it will call the *Send* function, which in turn calls the next .cpp (from the line above it in the conf file)

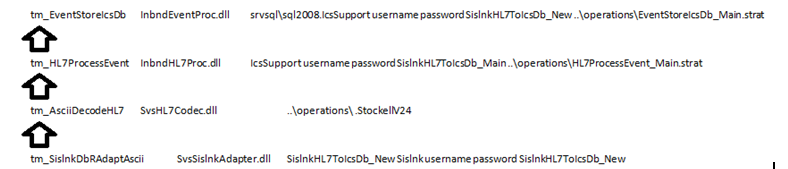
**

Figure 6



Figure 7

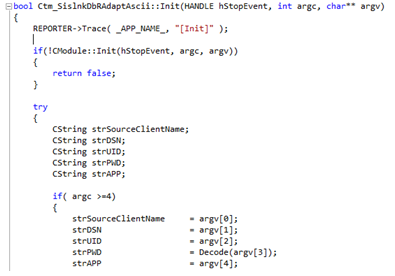
**

Figure 8

Additional notes:

* All of the above interfaces EXCEPT the IcsStmToIcsTrig have an associated conf file.
* To determine what cfg and conf files a given interface is using, bring *REGEDIT* and navigate to *HKEY\_LOCALE\_MACHINE -> SYSTEM -> CurrentControlSet -> services* and find the desired interface. Expand the interface, and click on *Parameters*. From there, you can see the cfg and conf path and file name.

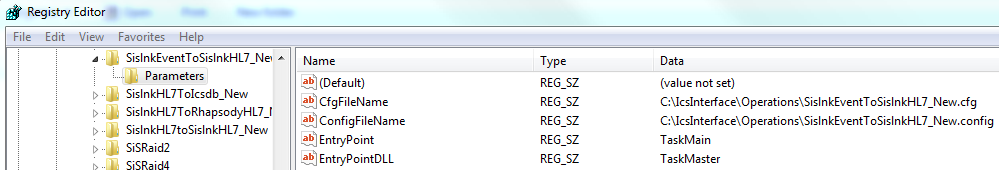
**

Figure 9

**Strats, MsgRules, and Segdefs**

Both the inbound and outbound interfaces use several other files to determine how to process messages. Those files are as follows:

* **Strategy Files** – Strategies are pre-defined rules, used by both inbound and outbound interfaces, which tell an interface how to process/create a given HL7 message. A strategy file (strat) contains a listing of each strategy that is being used by the interface.

The name of the strat file that a particular interface uses can be found in the .conf file. The strat file will have a *.strat* extension. Please see figure 6 above as an example.

Figure 10 below shows an example of a strat file. It consists of three parts - the strategy key, the dll name (solution) that it resides in, and the specific strategy that is being used. The specific strategy being used determines how we process certain elements on an HL7 message.

For example, as seen in Figure 10, there is a strategy key called VISIT\_STRATEGY, which is using the DefaultVisitStrategy. That specific strategy tells the interface to look at the PID.18 segment to find the visit number in an inbound HL7 interface message. However, we also have another strategy named PV1\_19VisitStrategy. That strategy tells the interface to look at PV1.19 for the visit number. So, if you would want to use that strategy instead, you would simply replace DefaultVisitStrategy with PV1\_19VisitStrategy in the file below.

So, as you can see, due to the large amount of strategies that are configured, there is much flexibility in how HL7 messages can be received/sent.

A complete listing of the strategies and their descriptions can be found in the [Interfaces Technical Design Doc](http://srvmoss/sites/intranet/Docs/Technical%20Design/Interfaces/Interfaces.doc).

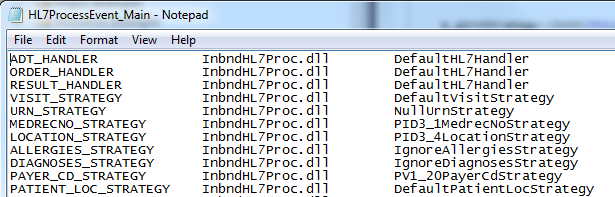
**

Figure 10

NOTE: The following interfaces are set up to use strategies: IcsTrigToSislnkEvent, SislnkEventToSislnkHL7, and SislnkHL7ToIcsDb.

* **MsgRules –** This file determines the segment order within an HL7 message that is allowable through the inbound and outbound interfaces. As shown below in figure 11, a rule is defined for each message type (A01, A02, etc…) that comes in (or goes out) through the interfaces. A given HL7 message must have its segments in the allowable order; otherwise the message won’t be processed, and the interface will go down.

In the example below, you will notice that the segment names are surrounded by either ‘[ ]’ or ‘{ }’. If a segment(s) is surrounded by ‘[ ]’, that entails that the segment(s) is optional. If a segment(s) is surrounded by ‘{ }’, that means that the segment(s) can be repeated. If there are no brackets around a segment, that means it is required. So, for example, below you will see *[{NK1}]* referenced several times. This means that the *NK1* segment is optional; if that segment is in the HL7 message, it can be repeated.

Also, as seen below, multiple segments can be enclosed within the brackets. For example, you will see the following: *[{IN1 [IN2] [IN3] [ZIN] }]*. This entails that that entire segment combination is optional, as it is surrounded by ‘[ ]’. Just inside the first open bracket and last closed bracket, you will see ‘{‘ and ‘}’. That entails that the entire combination of segments can be repeated. Then, by looking even deeper, you will see that the *IN2*, *IN3*, and *ZIN* segments each have ‘[ ]’ around them. That entails that each of those segments are optional within each combination instance.

So, in the above example, the following sequence of segments would be allowed:

IN1 IN2 ZIN IN1 IN3 ZIN IN1

However, the following sequence of segments would NOT be allowed, due to the fact that the 2nd IN2 segment does have an IN1 segment (which is required) before it.

IN1 IN2 IN3 ZIN IN2 IN3

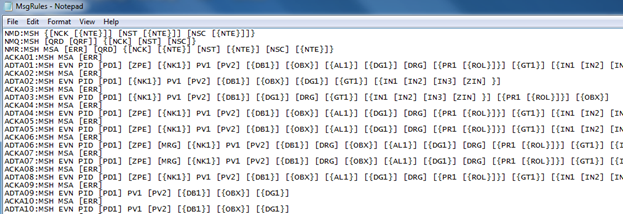
******

Figure 11

* **Segdefs –** Segment Definitions (segdefs) define the elements that are within each segment. Each segment that is present in the HL7 message needs to be defined in the segdef file. Figure 11 below shows an example of segment definitions for the *ACC*, *ADD*, and *AIG* segments.

In reading the Segdef file, the segment name can be found underneath its associated elements. For example, for the *ACC* segment, its name can be found under its 6 elements. Also, when reading the elements, they are in order from top-down. So, *ACC.1* refers to ‘Accident Date/Time’, *ACC.2* refers to the ‘Accident Code’, and so on.

For each element that the interface tries to write in a given segment, it must be defined within the segdef. If the interface attempts to write a given element that isn’t defined in the segdef, it will error out and stop the interface. For example, the *ACC*  segment has 6 defined elements in figure 12 below. If the interface attempts to create a message with a value in *ACC.7*, for example, it will error out because a 7th element is not defined in the segdef.

As far as the specific values on each line, they are not used by the interface. You really just need to make sure that a line exists for each element created by the interface.

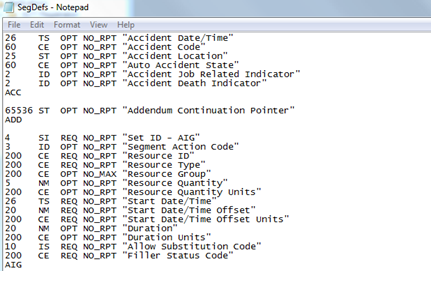
******

Figure 12

**Record Routing**

Our interfaces allow you to specify which message types you want to process and which ones you want to ignore. So, for example, if InsightCS generates an *A35* message, and the client chooses not to send that type of message outbound, you can tell the interfaces not to send it out. This configuration is defined in the *RECORD\_ROUTING* table in the Sislnk database.

Within that table, there are 3 fields – the *Source*, the *Destination*, and the *Record\_Type*. The *Source* and *Destination* fields refer to the specific interface *client\_num*, which is stored in the *CLIENTS* table. The *Record\_Type* field refers to the message type that you are wanting to process. The *Record\_Type* master records are defined in the *RECORD\_TYPES* table.

So, for example, if it is decided that *A05* messages should never make it to the *SislnkHL7ToEngineHL7* interface (and subsequently not get sent outbound), then the *A05* record type needs to be removed from the *RECORD\_ROUTING* table for that particular interface. To do that, you would first have to get the *CLIENT\_NUM* for the *SislnkHL7ToEngineHL7* interface. That value corresponds to the *DESTINATION* field in the *RECORD\_ROUTING* table. You would then need to determine the record type for the *A05* message in the *RECORD\_TYPES* table. Once you have that value, you would then simply remove the appropriate record in the *RECORD\_ROUTING* table.

The following query can be helpful in determining what message types are currently set up to be processed by a given interface.

select rr.source as 'source id',c.client\_name as 'source', rr.destination as 'dest id',c2.client\_name as 'destination',

rr.record\_type as 'record type',rt.description as 'description'

from record\_routing rr

inner join Clients c on c.client\_num = rr.source

inner join Clients c2 on c2.client\_num = rr.destination

inner join record\_types rt on rr.record\_type = rt.record\_type

where c.client\_name = 'SislnkEventToSislnkHL7'

order by c.client\_name, c2.client\_name

**Viewing an HL7 message**

Within the Sislnk database, each HL7 message is stored in the DATA column of the TRANSACTIONS table. You will notice that the data type for that column is an IMAGE. When you query that table, it will return hex characters. (see Figure 13 below)

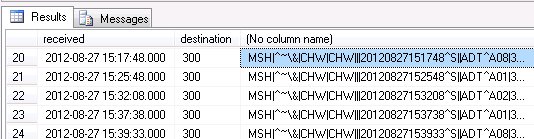


Figure 13

In order to be able to view the message through SQL, there is a ‘convert’ statement that will need to be run, which will convert the field into readable characters. (see below)

select received,destination, convert(varchar(8000),convert(varbinary(8000),data) ) from transactions

where received between '2010-09-10 00:00:00.000' and '2010-09-10 00:00:00.000'

**Common Issues Encountered in Support and Offshift and How To Troubleshoot them**

1. (Inbound) Insight is not receiving any inbound messages.
   1. Using the inbound engine log verify we sent an acknowledgment for the last message we received.
   2. Grab the port our inbound engine is listening on from its config file using monitor app.
   3. Open a command prompt on the machine and run the following command: netstat -an | findstr <port inbound engine is listening on>
   4. Verify the connection state is listening.
   5. If our inbound engine is listening attempt to restart the inbound engine.
   6. If messages still do not flow through it can be confirmed there is an issue with the foreign system.
2. (Outbound) Messages are not crossing to the foreign system.
   1. Look at our outbound engine’s log and verify we are attempting to send messages out.
   2. If we are look and see if we received any kind of acknowledgement. If we received an ack error, there most likely is some kind of special character the receiving system cannot process.
   3. If you do not see an ack verify we have an established connection with the foreign endpoint. Open a command prompt and run the following command: netstat -an | findstr <ip located in the outbound engines config file>.
   4. If the connection state is not established and we are just listening the foreign endpoint is most likely down and needs to be started again.
3. (Inbound) End of Wire difference is <a number>

This error means there is an issue with the message <a number> characters from the end of the message. This could mean the sending system did not send a carriage return after the message, there is an invalid character, or an invalid segment.

1. (Inbound) Received a batch of messages all at once.

Our inbound interfaces can only process one message at a time. This can be a result of the foreign system really attempting to send us a batch of messages due to an update they have recently made in their system or the sending system not waiting until we acknowledge a message before sending subsequent messages. Both of these errors will need to be resolved by the foreign system.