HL7v2 parsing

Introduction

HL7V2 was created prior to the advent of XML and established its own encoding formats to allow complex hierarchical data structures to be encoded into text strings. It was a significant advance on the previously standard fixed length data formats and was designed to allow backwards and forward compatibility. Unlike XML it does not add the field names to the message and prior out of band metadata is required to read the values in the message. This makes it less human readable than XML but results in very small messages which were originally designed to pass through 7 bit only transmission pathways. It is highly efficient but does require a careful parser implementation to take advantage of the format. These days XML parsers are common and widely available and jump start a developers ability to read a XML message, however this comes at a cost of message size. HL7V2 parsers are not difficult to write but a careful implementation is required and this technical report is designed to bring together the relevant information into one document to allow a high quality implementation of HL7V2 parsers.

HL7 Message Encoding

HL7V2 messages are most commonly ASCII text files with strict ASCII encoding (8 bit strings). They should only have cursor return (ASCII 13) and characters between ASCII 32 and ASCII 127 in the file. White space (the space character) is important. In more recent times some users have used Unicode or UTF-8 encoding of HL7V2 messages which avoids the need for escaping characters above ASCII 127 but requires special processing. These characters can be included in strict ASCII files by escaping the Unicode characters above the ASCII set, but in non-English speaking locales this can incur significant overhead as most characters need to be escaped. A unicode HL7V2 message should use a byte order mark (BOM) at the start of the file to indicate that it is unicode and can in general only be used with prior agreement. In Australia all standard HL7 Messages should use strict ASCII encoding as support of other encodings is very limited. For the purposes of this guide ASCII encoded messages are assumed.

HL7 Message Structure

HL7 Messages are collections of ASCII characters (between ASCII 32 and ASCII 127) separated by cursor return (ASCII 13) characters. Each string is called a segment and each segment is further divided into fields, which can optionally repeat, components and sub components. There is no ability to extend the hierarchy any deeper than this. However segments, while a flat list in the message can form their own hierarchies using the message structures detailed in the relevant chapters. This allows a further level of nesting to represent hierarchies.

A simple incomplete example of a HL7V2 message is below:

```
MSH|^~\&|MERIDIAN|Demo Server|||20100202163120+1100||ORU^R01|XX02021630854-
1539|P|2.3.1^AUS&&ISO^AS4700.2&&L||||AUS
PID|1||||SMITH^Jessica^^^^^L||19700201|F|||1 Test
Street^^WODEN^ACT^2606^AUS^C~2 Test Street^^WODEN^ACT^2606^AUS^C
```

This fragment of a message shows 2 segments, a "MSH" segment and a "PID" segment. It also shows fields, repeats of fields, components and sub-components. Theses are explained below.

To encode complex data into text, delimiters are required and these are specified after the MSH on the first line. In this case, which is the usual and in general only case in Australia, the delimitors are "|^~\&". HL7 Messages can only start with a "MSH" or a "FHS" and in both cases the delimiters are specified immediately after the MSH or FHS.

No field names appear in the messages and apart from the first 3 characters of every line and the delimiters after MSH/FHS all the text in a message is data. To read this data reliably requires knowledge of the HL7 standard and what position the data will appear in. The message can be reliably parsed without this knowledge but data extraction requires knowledge that resides in the standard itself. Specific knowledge of segments should **NOT** be used to parse messages as over time the segments will be extended and assumptions about the number of fields, or the presence or abscense of eg. sub components or repeats of fields will become incorrect and result in errors. It is common for a single text string (usually a "IS") data type to be extended to a "CE" data type in later versions and this is the most obvious example of why parsers should not assume the subcomponent structure in any field position. If the parsing is done without knowledge of the segment structure and data is extracted according to the HL7 guidelines then this type of change is backward and forward compatible.

Segments

Each line in a HL7 message is known as a Segment. It starts with a 3 character textual segment identifier which is always upper case. eg **"MSH"** or **"PID"** or **"PV1",** it ends at a ASCII 13 (cursor return) character. The three letter abbreviation is usually a mnemonic for its purpose. "MSH" stands for "Message Header" and PID for "Patient Identification". This will either be at the start of a message of immediately following a cursor return character (ASCII 13). Messages should terminate in a ASCII 13 character.

Fields, Components and Sub-Components

This fragment of HL7 is used to illustrate this:

```
PID|Field1|Component1^Component2|Component1^Sub-Component1&Sub-
Component2^Component3|Repeat1~Repeat2
```

- Segment - eg. PID
    - Field1 - Simple
    - Field2 - Has Components
        - Component1
        - Component2
    - Field3 - Components and Sub-Components
        - Component1
        - Component2
            - Sub-Component1
            - Sub-Component2
        - Component3
    - Field4 - Repeating field value
        - Repeat1
        - Repeat2

Each segment is divided into a variable number of fields. The number of fields depends on the version of the standard in use and how complete the data in the message is. No assumptions should be made about the number of fields in a given segment. Fields are separated by the field separator "|". The next level of the hierarchy is the ability of a field to repeat. This is indicated by the Repeat Delimiter "~" or tilde. No assumptions about the ability of a field to repeat should be made in parsing a message as a non repeating field can be made to repeat in later versions. Components are the next level and use "^" character as a delimiter. Components can be divided into Sub-Components using the "&" character. This is the limit of the hierarchy within HL7 V2 and all data structures must be encoded within these limits.

It is important to know that the delimiters are only added when required to separate data. In a segment with 30 fields, but only field 10 valued only the field delimiters prior to the value are included, all those after field 10 are omitted. As the number of fields in a segment varies with the version of a standard, trailing, unused delimiters serve no purpose and should ideally be removed to reduce message size. The same convention applies to Repeat, Component and Sub-Component delimiters.

Because these delimiter characters "|^~&" and the cursor return (ASCII 13) are used to define the tree structure of a HL7 message they must not appear within any text string in the data. To enable this all reserved characters are escaped using the "\" character (back slash). To add a delimiter character to the actual data contents the character is replaced by a special sequence of characters called an escape sequence. The escape sequence is always preceded and ended by the escape character "\".

The Escape Sequences are listed below. This applies to every field of every segment excluding the location in MSH and FHS segments where the delimiters are defined. Because the escape character itself may appear in a message it must also be escaped.

Field delimiter "|" "\F\"
Repeat delimiter "~" "\R\"
Component delimiter "^" "\S\"
Sub-Component delimiter "&" "\T\"
Escape delimiter "\" "\E\"
Cursor Return (ASCII 13) "\.br\"
Within all data in any place with a message this escaping must be done or else these reserved characters will drastically affect the message structure and result in potentially serious truncation or loss of data. This is not restricted to free text fields but applies to all data values in the message.

Within Formatted text fields other formatting characters are defined to allow the highlighting of text and control of page layout. These are covered later.

Datatypes

The HL7 Datatypes are documented in Chapter 2 of the HL7 International standard and in this chapter the way they are encoded using Components and subcomponents is documented. A Data type resides within a single field and does not use the repetition character, although the datatype itself may repeat.

The simplest data types are simple string values. Examples of this include "ST" and "IS". Any reserved characters with the string must be escaped but value is simply inserted into the message between 2 field delimiters as below.

```
An example ST Value: "|String Value|"
```

More complex types use Components and Sub-Components and this allows 2 two level hierarchy. Some data types such as the HD (Hierachic designator) is encoded using Component delimiters when it stands alone, but Sub-Component delimiters when embedded in another datatype such as a CX value.

```
An example HD Value (eg MSH Sending Facility):
"|Buderim GE Centre^7C3E3681-91F6-11D2-8F2C-444553540000^GUID|"
```

```
An Example XCN Value encoded with Another datatype (eg OBR Principle
result interpreter):
"|0191324T&McIntyre&Andrew&&&Dr.&&&AUSHICPR|"
```

```
Same XCN Value in a field by itself (eg OBR Ordering Provider):
"|0191324T^McIntyre^Andrew^^^Dr.^^^AUSHICPR^L^^^UPIN|"
```

Data types are also extended in later versions of the standard and no assumptions about the number of components or Sub-Components should be made.

Parsing HL7 V2

It is important to parse messages using the conventions above as this results in reliable access to data values even if the version of the standard is not what you expect. While it is possible to define a BNF grammar for HL7 V2 and parse messages using a generated parser, a hand coded parser is often used.

Messages should start with "MSH" or "FHS" and this sequence of characters is required to indicate the start of a message.

Once this is located the data can be split into segments by locating instances of the ASCII 13 character (Cursor return). The message should terminate with a ASCII 13 character.

No weight should be given to the segment name at this point and this process will result in an ordered list of segment strings.

The next step is to split these segment strings into the 3 character segment name and an ordered list of fields. This should be done by spliting the segment string using the field delimiter "|"

Each field should then be split into repeats using the Repetition Delimiter "~"

Each repeat is split into Components using the Component Delimiter "^"

Each Component is split into Sub-Components using the Sub-Component seperator "&"

At this point there is no need to take the escape character into account.

This process produces a tree of values (a parse tree), which for the example looks like this:

```
Example HL7 Fragment:
PID|Field1|Component1^Component2|Component1^Sub-Component1&Sub-
Component2^Component3|Repeat1~Repeat2
```

The resulting parse tree with values in parentheses:

- Segment = "PID"
  - F1
    - R1 = "Field1"
  - F2
    - R1
      - C1 = "Component1"
      - C2 = "Component2"
  - F3
    - R1
      - C1 = "Component1"
      - C2
        - SC1 = "Sub-Component1"
        - SC2 = "Sub-Component2"
      - C3 = "Component3"
  - F4
    - R1 = "Repeat1"
    - R2 = "Repeat2"

Legend
F Field
R Repeat
C Component
SC Sub-Component
A tree has leaf values and nodes. Only the leaves of the tree can have a value. All data items in the message will be in a leaf node.

At this point the data items in the message are in position in the parse tree, but they can remain in their escaped form. To extract a value from the tree you start at the root of the Segment and specify the details of which field value you want to extract. The minimum specification is the field number and repeat number. If you are after a component or sub-component value you also have to specify these values.

If for instance if you want to read the value "Sub-Component2" from the example HL7 you need to specify: Field 3, Repeat 1, Component 2, Sub-Component 2 (PID.F1.R1.C2.SC2) Reading values from a tree structure in this manner is the only safe way to read data from a message and is very fast for accessing values in most implementation environments.

All values should be accessed in this manner. Even if a field is marked as being non-repeating a repeat of "1" should be specified as later version messages could have a repeating value.

To enable backward and forward compatibility there are rules for reading values when the tree does not match the specification (eg PID.F1.R1.C2.SC2) The common example of this is expanding a HL7 "IS" Value into a Codeded Value ("CE"). Systems reading a "IS" value would read the Identifier field of a message with a "CE" value and systems expecting a "CE" value would see a Coded Value with only the identifier specified. A common Australian example of this is the OBX Units field, which was an "IS" value previously and became a "CE" Value in later versions.

```
Old Version: "|mmol/l|" New Version: "|mmol/l^^ISO+|"
```

Systems expecting a simple "IS" value would read "OBX.F6.R1" and this would yield a value in the tree for an old message but with a message with a Coded Value that tree node would not have a value, but would have 3 child Components with the "mmol/l" value in the first subcomponent. To resolve this issue where the tree is deeper than the specified path the first node of every child node is traversed until a leaf node is found and that value is returned.

This is a general rule for reading values: **If the parse tree is deeper than the specified path continue following the first child branch until a leaf of the tree is encountered and return that value (which could be blank).**

Systems expecting a Coded Value ("CE"), but reading a message with a simple "IS" value in it have the opposite problem. They have a deeper specification but have reached a leaf node and cannot follow the path any further. Reading a "CE" value requires multiple reads for each sub-component but for the "Identifier" in this example the specification would be "OBX.F6.R1.C1" The tree would stop at R1 so C1 would not exist. In this case the unsatisfied path elements (C1 in this case) can be examined and if every one is position 1 then they can be ignored and the leaf of the tree that was reached returned. If any of the unsatisfied paths are not in position 1 then this cannot be done and the result is a blank string.

This is the second Rule for reading values: **If the parse tree terminates before the full path is satisfied check each of the subsequent paths and if every one is specified at position 1 then the leaf value reached can be returned as the result.**

In the second example every value that makes up the Coded Value, other than the identifier has a component position greater than one and when reading a message with a simple "IS" value in it, every value other than the identifier would return a blank string.

Following these rules will result in excellent backward and forward compatibility. It is important to allow the reading of values that do not exist in the parse tree by simply returning a blank string. The two rules detailed above, along with the full tree specification for all values being read from a message will eliminate many of the errors seen when handling earlier and later message versions.

At this point the desired value has either been located, or is absent, in which case a blank string is returned. To return the value to the requester however this returned string must be checked for escape characters and these characters dealt with. This is detailed in the next section.

Dealing with reserved characters and delimiters

As detailed above the HL7V2 delimiters cannot appear in the data of a message or they would distort the data tree and cause serious data loss. They will be escaped using the escape sequences detailed above. Free Text also has formatting commands with there own escape sequences, but every field must handle the case of delimiters.

Once a value have been extracted from a parse tree it must be checked for escape characters and if present these escape sequences need to be converted back into the unescaped version. Generally this involves replacing a sequence of characters with a single character.

The backslash "\" is the escape character in the string and if there are no backslash characters in the string nothing needs to be done. However if there are any backslash characters in the string these must be dealt with. It is important to note that this must not be done with search and replace as this will yield erroneous results in many cases.

The only reliable way to remove escape characters is to iterate over the string from the start testing for the presence of a backslash ("\") character. If one is found then the following characters, up to the next "\" are read and this value looked up and the whole sequence, including the 2 backslash characters replaced with the correct character.

Example 1: the input string of "10\S\9/l" becomes "10^9/l"

Example 2: the input string of "Obstetrician \T\ Gynaecologist" becomes "Obstetrician & Gynaecologist"

Example 3: the input string of "201104\E\123456" becomes "201104\123456"

Free Text Formatting

In free text fields there are a large number of other formatting commands that may appear which are enclosed in escape sequences. These must be converted to an appropriate formatting code in the destination formatting scheme. Because they are enclosed in escape characters it is often efficient to handle them when unescaping the delimiter characters, but they can be handled in other ways as long as search and replace is not used, as this will cause errors depending on the order in which the search and replace is executed.

Unicode characters

The HL7 standard describes multi-byte character set escape sequences to allow re-selection of character set code pages that are often used for languages that cannot use the ISO 8859 codepage. eg Japanese Cyrillic. These have have been superceded by Unicode which provides a single character encoding to encompasses all languages. Unicode should be used only by site agreement, and is not recommended for general use as limited support is available in receiving systems. Character set selection escape sequences (\Cxxyy\ and \Mxxyyzz\ should not be used). In the case where Unicode characters are required for display purposes, an xHTML display segment in ASCII encoding is suggested. In this encoding Unicode characters can be represented in ASCII xHTML using the HTML escape sequences for Unicode characters. The use of any characters greater than 127 is not allowed.

Example: the cent sign (¢) is represented using "¢" in the html. In the actual message HL7 escaping would result in "\T\#162;"