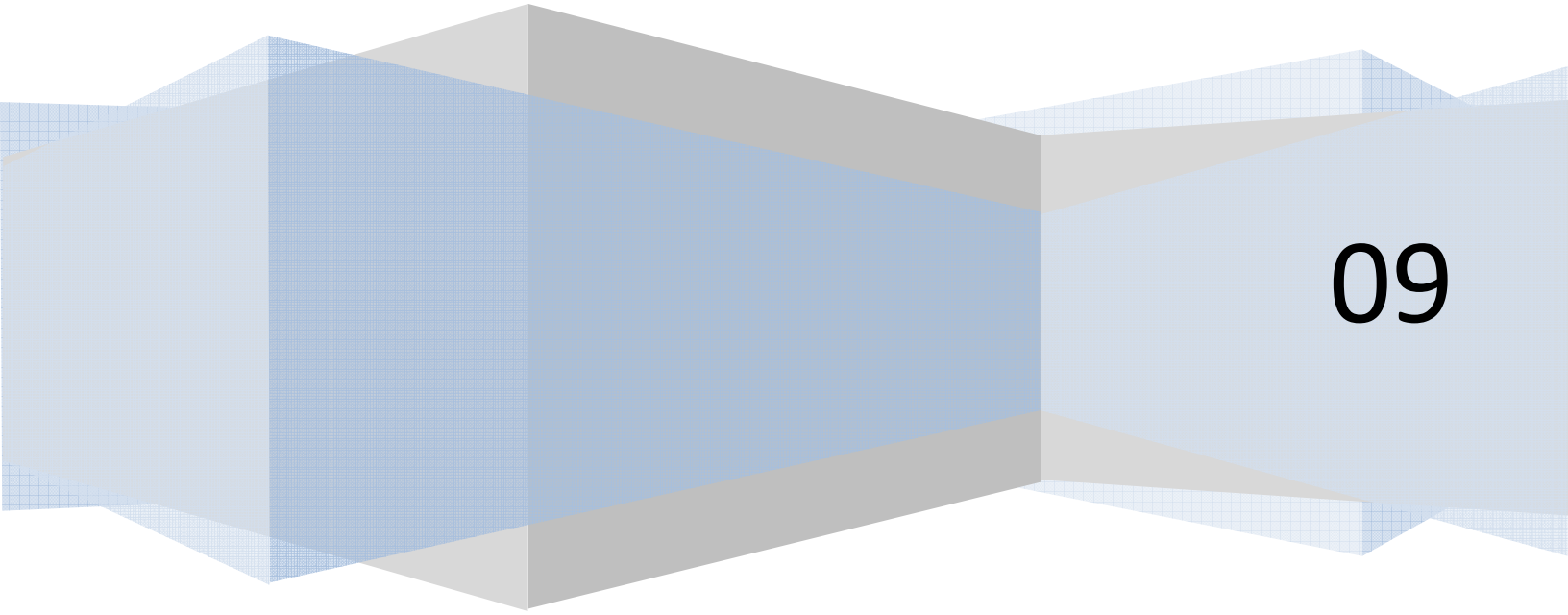
BonCode (www.BonCode.net)

Fixed File Reader

How to use the Fixed File Reader component to parse complex fixed files.



09

Table of Contents

Fixed File Reader

License

The Beginning

So you get into a situation where someone throws a file at you saying: “We need to read this information and it should be easy since it is fixed file.”

It looks like this (added column and row numbering for visualization only):

```
        123456789012345678901234567890123456789012345678901234567890123456789
1       JOHN      DOE       ATLANTA    GA
2       BRIAN     STERLING  MEMPHIS    TN
3       SAMANTHA  ROCKFORT  WASHINGTON DC
```

Example 1

Built in Solutions

You think no problem, this could be easy; I will just ask them to make a small change and resend it in some sort of delimited format. This will allow me to use a built in tag like <CFHTTP>. Never quite understood why the ability to parse delimited file ended up in this tag instead of <CFFILE> tag, but I guess that is old history.

Unfortunately, most of the time this will not happen and you will either use something like a platform text driver, or have to parse the file yourself writing specific parsing code. For a tabular layout like the one in Example1, you could use the windows text driver (http://msdn.microsoft.com/en-us/library/ms709353(VS.85).aspx).

The Problem Situation

However, all this breaks down when we deal with more complex fixed files, which contain interlocked data relationships and may change meaning of data with every line, a good example for this is the Visa Commercial Format (https://usa.visa.com/corporate/corporate_solutions/im/vcf/contact.jsp). After dealing with yet another complex fixed file, and having to write yet another program to read it, I got to thinking that there had to be a better and easier way. Of I went to the trusted source of all knowledge, Google. However, after much googling I came away empty handed. So, I decided to see whether I can

create something that can handle these files and does not require a complete re-invention of the wheel every time.

Example 2 shows a fictitious fixed file which contains information about parents and their children. It contains three couples. John and Carla, have a daughter (Samantha) and a son (George); Peter and Patricia who have a son, Stuart, and Allen and Maria, who have a daughter, Gonzuela. There are several other data points that we can explore. Each parent line has an indicator of "P" as the first character, while "D" and "S" indicators are used for the children; they distinguish daughters and sons. The number of children for each couple is also indicated on the parent lines on positions 42-43. And finally each child has listed their favorite action figure or doll.

```
        12345678901234567890123456789012345678901234567890123456789
1       P JOHN       DOE       CARLA     DOE       02
2       D   SAMAMTHA        DOE       BARBIE
3       S   GEOERGE         DOE       G.I. JOE
4       P PETER      MAYE      PATRICIA  MAYE      01
5       S   STUART          MAYE      BATMAN
6       P ALLEN      GONZALES MARIA     VALENCIA  01
7       D   GONZUELA        GONZALES  BARBIE
```

Solution

The solution to be able to parse the above example and more complex ones presented itself for me in two parts. Part 1 is core library, in my case a cfc (FixedFileReader.cfc). Part 2 is a definition file. So, while the core library is not expected to change, the definition file will contain all the changes describing the differences in the type of fixed file to load. This implementation has not been optimized for speed, though I have been able to parse files up to 2MB without any issues. Also, fixed file scenarios can become fairly complex and thus this particular reader may not work for you and you still will need to write your own parser.

Usage

The coding aspects of the Fixed File Reader are fairly straight forward. You initialize the component, you set two properties, and off it goes. What is requiring some trial and error is the definition file; so most of the usage information is going to focus on the options in the definition file.

```
<cfscript>
      objFFR = CreateObject("component","FixedFileReader");
      //set debug option to on or off
      objFFR.setDebug(false);
      //now activate parser and provide content file and definition file
      stcResult = objFFR.fReadFixedFile("c:\Content.txt","c:\Definition.xml");
      //if we need additional parsing done we need to call the init() function
      objFFR.init();
</cfscript>
```

ResultArray: An array of structures. The ResultArray contains an index for every line of the fixed file to be parsed. If your fixed file had 272 lines, then the size of your ResultArray will be 272 as well. Each index, in turn contains a structure reflecting the fields and data of the line that was parsed.

ResultQueries: Depending on your file definition some or all data may be returned as queries. The query columns will correspond to the <field> definitions. There are only two data types supported at this point for query columns: numeric and text.

ResultXML: This is for future implementations and should always be an empty string for now.

intErrLines: The number of lines that were parsed with errors. These are normally type conversion errors.

ProcessingTime: The time in seconds the parser required to parse the file.

There are also three fields that are introduced by the fixed file parser (ParserRecordID, ParserParentID, and ParserMatchCase). You cannot use these in your field names, i.e. these are reserved names. These will be present in the ResultQueries and in structures within ResultArray. ParserRecordID indicates on which line of the file the data was found, while ParserParentID indicates the parent line if relationships were used, and ParserMatchCase will contain the <match> tag condition that was used to parse the line and determine field definitions used.

The Basic Definition

The definition file is an XML file thus it has root elements, nodes (tags), and attributes. The basic definition construct consists of :

```
<boncodeFixedFileDefintion …>
      <header …>
            <match …>
                  <field …> </field>
                  <header …>
                        …
                  </header>
            </match>
      </header>
</boncodeFixedFileDefintion>
```

<boncodeFixedFileDefintion> can contain child <header> nodes. <header> nodes, in turn, can contain <match> nodes, which can contain additional nested <header> nodes as well as <field> nodes.

The root element determines only how the fixed file terminates each line. Thereafter the header/match/field combinations are used to parse data. You use the header node to declare a line that may have additional lines in the same file that can be attributed or related to it, i.e. children. Using our Example 2, lines 1, 4, and 6 would qualify to be headers. The fixed file from Example 1, is an extreme example, where you would have headers but no children. Within each header you, then, determine how to interpret the header's data by declaring <match> nodes. Each <match> node can contain <field>

nodes to determine how data is extracted in each line. If a given line can have sub-nodes, e.g. children of children (grandchildren), then additional headers can be nested within them to determine how to interpret subsequent lines of data.

The order in which nodes are listed matters, as we will evaluate tags in order of declaration.

We will work through all attributes for each of the nodes, then, look at constructing full-fledged definitions for some of our samples.

< boncodeFixedFileDefintion >

The root node only has one valid attribute.

Example:

<boncodeFixedFileDefinition EndOfLine="**CRLF**">

Attribute	Description
EndOfLine	Optional: Determines how the fixed file terminates each line. Defaults to the CRLF (Carriage Return, Line Feed) combination. An alternate character for line termination may be supplied. The following are interpreted line terminators that can be used instead of the character codes: CR = Carriage Return (0x0D / 13) LF= Line Feed (0x0A / 10) CRLF = a combination of CR and LF TAB = tab character (0x09/9)

< header >

The header node has several attributes. Several nodes (tags) allow the use of expression attributes. In such cases, valid ColdFusion functions may be used to operate on line data. The line data is represented by a question mark (?). For example, you may see childExpression with “Function:Left(?,1)”. Given line data of Example 2 above, this would return the character “P” for the first line, the character “D” for the second line and so on.

Example:

<header name="**Root Header**" repeat="**Yes**" IgnoreLastRecordData="**yes**" restartAtLastRecord="**Yes**" childExpression="**Function:Left(?,1)**" endExpression="**Function:Left(?,1) IS 9**">

Attribute	Description

Name	Optional: the name for the header. This will be used in debug output if provided
repeat	Optional: Yes/No: Once the end of the header is found, should the reader look for another occurrence of this header? Default=false. Most likely the root header has the repeat option turned on, while child headers have this set to false.
repeatUntil	Optional: if repeat is enabled, it can be unconditional (default) so that it always repeats, or a condition can be specified in the repeatUntil attribute.
IgnoreLastRecordData	Optional: Yes/No: When the end of the header is detected, the line on which we detect the end condition may require specific treatment. It may also be formatted differently. Setting this directive to Yes will set the row data to a string, rather than a structure.
restartAtLastRecord	Optional: Yes/No: If we have found the last line of a header, we can reprocess the stop line with the parent's <match> definition. If this is desired, this should be set to "Yes". If "No" the header recognition will pick up with the next line (row).
childExpression	Required: This determines how we recognize children via the <match> tags. You specify this as [Type:Content]. Can be one of two types, Function or ParentField: **Function:** A function can be any valid ColdFusion function, where the line content can be substituted by question mark. For example, setting the Attribute to "Function:Left(?,1)" will retrieve the first character of the line for further evaluation by the <match> tag. **ParentField**: Except for the first <header> node, all subsequent nodes can make use of the ParentField concept. This will retrieve data from the already parsed parent fields; all fields that have been parsed by the parent line are available, e.g. childExpression="ParentField:RecordTypeCode". This will retrieve the RecordTypeCode field that is assumed to be one of the fields parsed on the parent line. We can use it to define different use of field definition, e.g. if RecordType contained either "Pets" or "Toys" we can use different field definition to either capture toy fields or Pet fields. We do so by declaring appropriate <match> tags.
endExpression	Required: This determines how we recognize when a <header> tag is complete, i.e. all related data has been processed. You specify this as [Type:Content]. Can be one of two types, Function or Count: **Function:** A function can be any valid ColdFusion function; it has to evaluate to true or false. If true we have reached the end of the <header> definition. The next line will be using a different header definition, or if we have chosen the repeat attribute, we start again using the current header. **Count**: Count can be either numeric or reference to a ParentField. For example, "Count:5" would mean that we have 5 rows for the current header's children. Similarly "Count:childCount" would mean we would look at the childCount field of the parent record and use its content to determine how many rows are associated with the <header> .

< match >

The match node uses information defined in the childExpression attribute to determine whether and what fields to parse out of a given line. Multiple <match> tags may be used within a <header> tag.

Examples:

<match case="**8**" query="**vcfHeader**" name="**vcf segment header record**" segmentDel="**tab**">

OR

<match case="**D**" name="**Daughter Line**">

OR

<match case="**02**" name="**Type 02: Account Header Type**">

Attribute	Description
case	Required: when evaluating the child expression of the header, we will only choose to use the <field> definitions within this <match> if the evaluation for the childExpression is equal to the case specified. For example, if the <header> childExpression is “Function:Left(?,1)” and the expression returns a “D”, then, corresponding <match> tag to process this should have a case attribute of “D” as well.
name	Optional: the name for the match. This will be used in debug output if provided.
query	Optional: The name of the query to which the parsed fields should be added. Multiple <match> tags may refer to the same query. In such a case the query will be appended as needed.
segmentDel	Optional: If the line contents are delimited by a character rather than field size, you can specify a segment delimiter. The parser will interpret the following delimiters declarations automatically to the correct characters: TAB = 0x09 COMMA = 0x2C/44 If segmentDel is used, the use of the <field> nodes are optional. The parser will assign numeric values to fields starting from “Field000”. Also if you only provide partial <field> definition, the parser will assign numeric field definitions for the remainder of the fields using a similar naming scheme.
Ignore	Optional: yes or no. We can add match condition that we do not want to process any further for completeness. If so, we can direct the parser to ignore the match, this will result in slightly different result. Instead of

	undefined array element, you will see a string in the returned array for the array index.

< field >

Once the processing of a line has identified the appropriate <match> tag to use, the further definitions of what data is to be parsed is determined via the <field> nodes (tags). The <field> nodes are optional for <match> nodes that have a segmentDel attribute specified, but nonetheless for readability purposes and more informative definitions they should still be used.

Even with fixed size fields, you will only need to supply a limited set of <field> nodes up to the point of data that you will need. The parser will ignore data to the end of the line if field definitions are missing.

Examples:

```
<field order="1" name="DateOfCharge" type="date" />
<field order="2" name="ItemSequenceNumber" type="numeric" />
<field order="3" name="RoomRate" type="numeric" format="Function:DIV100" />
```

OR

```
<field order="1" count="14" type="numeric" format="Function:DIV100"
name="TransactionAmount" />
```

OR

```
<field order="2" count="22" type="text" name="AccountCode" />
```

Attribute	Description
name	Required: name of field. This has to be a valid ColdFusion variable construct, i.e. no spaces or special characters except underscore. The data will placed in a structure key using this key name
count	Required: Required only required when we do not have a segmentDel attribute in the corresponding <match> node. This is count of characters that the field occupies. Thus, the order of <field> nodes is important. The first node is assumed to start with the first character, then the count is added to determine the end of the field. The second field starts with the next character on the line and the appropriate “count” is added to determine its end and so on.
type	Optional: The expected data type of the field. Currently three data types are supported: text, numeric, and date. The default Is “text”
format	Optional: the numeric and date data type allow for additional formatting to be supplied. Currently only the formatting options for the numeric data type have been implemented.

	The following are implemented format functions: DIV100: Divides the number in the file by 100 DIV1000: Divides the number contained in fixed file by 1000.
order	Optional: this is not used in processing, but can assist in visual layout of the file to keep order of the fields visible.

That is all there is to it.

Walk Through Example

Now we can use the information to look at our data from Example 2 again and see how we can write definition files to parse the data. There are several ways we can define headers, especially the endExpressions can have some flexibility. I have chosen to make use of the count of children that is on each parent record to let the parser know when to stop looking for additional child records.

We use the following definition file to parse our Example2 data: (copy and paste to notepad to see full indentation more easily):

```
<!-- Defition for Reading Example file 2 -->
<boncodeFixedFileDefintion EndOfLine="LF">
        <!-- the repeat attribute tells to parser to go look for the next
header segment once we found the end of the existing one -->
        <header name="Parent Header" repeat="Yes" IgnoreLastRecordData="no"
restartAtLastRecord="Yes"
               childExpression = "Function:Left(?,1)"
endExpression="Function:Left(?,1) IS 'P'"> <!-- end of a segment (group of
multiple records) can be denoted via count (e.g. count:13 or via a Function -
->
                <!-- match elements for case can be any alpha numeric combo
but cannot contain comma -->
                <match case="P"  name="Parent Match"> <!-- if you provide
segmentDel attribute the count attribute for fields will be ignored, you can
use tab,comma keywords -->
                        <field name="RecordType" count="2" />
                        <field name="Father_FirstName" count="10"/>
                        <field name="Father_LastName" count="9"/>
                        <field name="Mother_FirstName" count="10"/>
                        <field name="Mother_LastName" count="10"/>
                        <field name="ChildCount" type="numeric" count="2"/>

                        <!--- check for children -->
                        <header name="Child Header" IgnoreLastRecordData="no"
restartAtLastRecord="Yes"
                                childExpression = "Function:Left(?,1)"
endExpression="Count:ChildCount">
```

```
                    <!-- find daugthers -->
                    <match case="D"  name="Daughter Match">
                        <field name="ChildType" count="4"/>
                        <field name="Daughter_FirstName"
count="15"/>
                        <field name="Daughter_LastName"
count="10"/>
                        <field name="Daughter_FavoriteDoll"
count="10"/>
                    </match>
                    <!-- find sons -->
                    <match case="S"  name="Son Match">
                        <field name="ChildType" count="4"/>
                        <field name="Son_FirstName"
count="15"/>
                        <field name="Son_LastName"
count="10"/>
                        <field name="Son_FavoriteActionFigure"
count="10"/>
                    </match>
                </header>
            </match>
        </header>
</boncodeFixedFileDefintion>
```

Figure 4: Definition file for example 2

Dumping the result structure for Example 2 would look like this (note the Parent References on each line):

struct

INTERRLINES	0
PROCESSINGTIME	0.219
RESULTARRAY	**array**

1 **struct**

ChildCount	2
Father_FirstName	JOHN
Father_LastName	DOE
Mother_FirstName	CARLA
Mother_LastName	DOE
PARSERMATCHCASE	P

	PARSERPARENTID	0
	RecordType	P
2	**struct**	
	ChildType	D
	Daughter_FavoriteDoll	BARBIE
	Daughter_FirstName	SAMAMTHA
	Daughter_LastName	DOE
	PARSERMATCHCASE	D
	PARSERPARENTID	1
3	**struct**	
	ChildType	S
	PARSERMATCHCASE	S
	PARSERPARENTID	1
	Son_FavoriteActionFigure	G.I. JOE
	Son_FirstName	GEOERGE
	Son_LastName	DOE
4	**struct**	
	ChildCount	1
	Father_FirstName	PETER
	Father_LastName	MAYE
	Mother_FirstName	PATRICIA
	Mother_LastName	MAYE
	PARSERMATCHCASE	P
	PARSERPARENTID	0
	RecordType	P
5	**struct**	
	ChildType	S
	PARSERMATCHCASE	S
	PARSERPARENTID	4
	Son_FavoriteActionFigure	BATMAN
	Son_FirstName	STUART
	Son_LastName	MAYE
6	**struct**	
	ChildCount	1
	Father_FirstName	ALLEN
	Father_LastName	GONZALES
	Mother_FirstName	MARIA
	Mother_LastName	VALENCIA
	PARSERMATCHCASE	P

	PARSERPARENTID	0
	RecordType	P
7	**struct**	
	ChildType	D
	Daughter_FavoriteDoll	BARBIE
	Daughter_FirstName	GONZUELA
	Daughter_LastName	GONZALES
	PARSERMATCHCASE	D
	PARSERPARENTID	6
RESULTQUERIES	**struct [empty]**	
RESULTXML	[empty string]	

Figure 5: Dump of Result for Example 2

Now we will make a small change to the parser definitions and ask for queries to be returned for matching daughters. We do so by adding the query attribute to the <match> node:

<match case="**D**" query="**selDaughters**" name="**Daughter Match**">

The structure key ResultQueries will be returned with all daughters from all parents put into one query:

struct

selDaughters — **query**

	CHILDTYPE	DAUGHTER_FAVORITEDOLL	DAUGHTER_FIRSTNAME	DAUGHTER_LASTNAME	PARSERMATCHCASE	PARSERPARENTID	PARSERRECORDID
1	D	BARBIE	SAMAMTHA	DOE	D	1	2
2	D	BARBIE	GONZUELA	GONZALES	D	6	7

Figure 6: Dump of Result as Query

Note that the ParserParentID and ParserRecordID maintain the correct references. This assists in relating data among multiple queries that can be returned.

Using Debugging

The parser can work in debug mode by setting the “setDebug(true)” option. This is a very verbose presentation of the internals of the processing as it occurs. The following would be an example to illustrate the debug output for parsing the Example 2 parent/child data.

```
FixedFileReader started at timestamp {ts '2009-12-22 10:31:33'}
Element counts will not include parser defined fields PARSERMATCHCASE and PARSERPARENTID.

Starting Header [Parent Header] at line 1
Match for [Parent Match] on condition [Function:Left(?,1)] to [P] for line 1
-- parsed 6 elements
Starting Header [Child Header] at line 2
Match for [Daughter Match] on condition [Function:Left(?,1)] to [D] for line 2
-- query object [selDaughters] created.
-- parsed 4 elements
Match for [Son Match] on condition [Function:Left(?,1)] to [S] for line 3
-- parsed 4 elements
-- used count indicator showing 2 records
End of header detected using expression [Count:ChildCount] at line: 3
End of header detected using expression [Function:Left(?,1) IS 'P'] at line: 1
Starting Header [Parent Header] at line 2
Match for [Parent Match] on condition [Function:Left(?,1)] to [P] for line 4
-- parsed 6 elements
Starting Header [Child Header] at line 5
Match for [Son Match] on condition [Function:Left(?,1)] to [S] for line 5
-- parsed 4 elements
-- used count indicator showing 1 records
End of header detected using expression [Count:ChildCount] at line: 5
End of header detected using expression [Function:Left(?,1) IS 'P'] at line: 4
Starting Header [Parent Header] at line 5
Match for [Parent Match] on condition [Function:Left(?,1)] to [P] for line 6
-- parsed 6 elements
Starting Header [Child Header] at line 7
Match for [Daughter Match] on condition [Function:Left(?,1)] to [D] for line 7
-- parsed 4 elements
-- used count indicator showing 1 records
End of header detected using expression [Count:ChildCount] at line: 7
End of header detected using expression [Function:Left(?,1) IS 'P'] at line: 6
Starting Header [Parent Header] at line 7

FixedFileReader ended at timestamp {ts '2009-12-22 10:31:33'}
```

Figure 7: Debug sample

Summary

The fixed file reader component can be used to read complex fixed files. For simple tabular layouts there are faster platform specific solutions. Even for complex layouts, it may not be appropriate for all situations. The hardest part will be to get your definitions right. You will have to experiment with it a little bit and do some iterative development. I would suggest you turn of the “repeat” options and define one header at a time.

I have included several more complex definitions for review, including one that shows how to parse an EDI X.12 PO.

Please leave comments on the bulletin boards if you wish to provide feedback.

Best of luck,

Bilal