Quick Start Guide for FHiCL 3: The **F**ermilab **Hi**erarchical **C**onfiguration **L**anguage

Walter E. Brown, Chris Green, Kyle Knoepfel, Jim Kowalkowski, Marc Paterno, Ryan Putz
Fermilab/SCD/SSA/SSI
Revision 4

Contents

1 Introduction

The purpose of this document is to explain and demonstrate the syntax and semantics of the Fermilab Hierarchical Configuration Language, FHiCL[1], so that users may become comfortable with its features and intended use.

The end product of many years of experience with other configuration languages and notation, FHiCL has been carefully designed to allow its users to express, record, and retrieve sets of software parameters used to *configure* (prepare for a particular purpose) a program's execution. We will use the term *parameter set* to denote any specific collection of named values accessible to a user's program while it is running; a FHiCL *document* is a textual representation of such a parameter set. Two or more FHiCL documents are said to be *equivalent* if they lead to identical parameter sets.

Depending on the programming language used to write the user program, one of several software subsystems is used to produce a parameter set from a given FHiCL document. Each such subsystem is known as a *binding* of the FHiCL specification to the programming language, and provides for the analysis and interpretation of FHiCL documents.

[1]The customary pronounciation is /fIkl/.

While it is up to each binding to specify how users interface to the parameter sets produced by that binding, it is fundamental that users be able to query a parameter set by providing it a name (in the form of a string) in order to obtain the value corresponding to that name. Further, a parameter set must be capable of producing a document that is equivalent to the original document that gave rise to that parameter set.

2 Documents

2.1 Text

A FHiCL document is simply a sequence of characters (*i.e.* text), structured as described below, and is commonly stored in a file whose name is conventionally suffixed **`.fcl`**. For example, **`my_config.fcl`** might be the name of such a file. Any conventional text editor (*e.g.* emacs, vi, nedit, . . .) may be used to create or to update a FHiCL file.[2]

2.2 Name-value pairs

A document consists principally of *name-value pairs*.[3] There may be as many or as few such pairs as desired.[4] In each pair, a colon (`:`) separates the *name* from the corresponding *value*.[5]

At least one blank, tab, or newline character (collectively known as *whitespace*) must separate one pair from the next. The following document, consisting of three name-value pairs, uses the minimum required whitespace:

```
1 n:1 pi:3.14159 label:"horizontal axis"
```

2.3 Optional whitespace

Within a FHiCL document, additional whitespace may be used at the discretion of the author.[6] A binding will ignore any such optional whitespace while producing a parameter set from the document.

---

[2] Note that FHiCL documents need not be represented via any file. A binding may, for example, obtain a FHiCL document from a database, via a string in a conventional programming language, or via any other mechanism that can denote simple text. A binding is free to support an arbitrary number of document sources.

[3] A name-value pair is sometimes known as an *association*, because parameter set lookup is designed to take a name and retrieve the associated value. In this Guide we will usually prefer the simpler *pair* nomenclature.

[4] A document consisting of no name-value pairs is said to be *empty*, as is the parameter set that a binding would generate from such a document. Both the empty document and the empty corresponding parameter set are valid FHiCL constructs.

[5] In the context of the FHiCL language, the colon is referred to as the *standard binding operator*.

[6] Such optional whitespace is commonly used to produce indentation or alignment.

The following document is equivalent to the single-line document shown above. Consisting of the same three name-value pairs, this variation employs extra whitespace to improve readability by (a) placing each pair on an individual line and (b) aligning the values:

```
1 n     : 1
2 pi    : 3.14159
3 label: "horizontal axis"
```

3 Comments

Document providers often wish to annotate the document's contents. For example, it is common to provide a provenance giving the original author's name and date, followed by a revision history. Other annotations include introductory overviews for each section of a document, or even brief descriptions of individual name-value pairs.

FHiCL provides two ways to *introduce* (start) a comment:

- With a single **#** character,[7] or else
- With two consecutive forward slashes (**//**).[8]

Either or both of these comment introductions may be used within any FHiCL document.

The remainder of the line is the *body* of the comment, and provides whatever information the author may desire. The comment implicitly terminates at the end of the line, although an author may choose to continue his annotation onto any number of additional comments on subsequent lines.

The following document illustrates the various forms that a FHiCL comment may take.

```
1 # This is a comment
2 // This is also a comment
3 foo : "bar"   # this is a comment "in the margin" ...
4 foo2: "bar2" // ... and so is this
```

4 Names

4.1 Spelling

The spelling rules for FHiCL names match the spelling rules for identifiers in many programming languages:

- Each name begins with a letter or with a _ (underscore) character.
- The name may be spelled with as many additional consecutive letters, underscores, or digits as desired. No other characters (*e.g.* punctuation or whitespace) may be embedded within a name.

[7] This notation is adopted from such scripting languages as bash, perl, and python. The introductory **#** character is known variously as a *pound sign*, *hash mark*, *sharp*, or *octothorpe*.

[8] This notation is adopted from such programming languages as BCPL and C++.

- Capitalization matters: the name **`X`** is not the same name as the uncapitalized name **`x`**. Similarly, FHiCL treats the names **`Hello`**, **`HELLO`**, and **`hello`** as three distinct, unrelated names.

4.2 Name reuse

Consider the following document, noting especially the reuse of the name **`a`**. When a binding processes this document, how many name-value pairs will the resulting parameter set contain?

```
1 a: 1
2 b: 2
3 a: 3
```

The answer is two. FHiCL provides that, if a document has two name-value pairs that have a name in common, the value in the later pair *overrides* (supercedes) the earlier one.[9] Therefore, in the above example, when **`a`** is looked up in the parameter set, the associated value will be found to be **`3`**. Section 7 discusses how the values associated with previously defined names can be used in other locations of the document.

5 Values

5.1 Classifications

At a high level, each FHiCL value can be classified as either *atomic* or *structured*. A structured value can further be categorized as a *sequence* or a *table*. Each value, therefore, falls under one of the following three FHiCL *categories*:

atom: A value that has no underlying structure.
sequence: A collection of values that are not associated with any names.
table: A collection of name-value pairs.[10]

We first discuss values of atomic type, and then discuss the sequence and table.

5.2 Atomic values

5.2.1 Boolean values

The literals **`true`** and **`false`** correspond to the customary truth values.[11]

```
1 debug: true
```

[9] If more than two pairs have a name in common, the second pair overrides the first as described above until the third such pair is encountered. Then the third pair overrides the second until a fourth pair is encountered, and so on. In this way, the last pair using that name will ultimately override all the earlier ones with the same name.

[10] Also commonly referred to as a parameter set.

[11] Any language binding will convert these literals into the Boolean representations native to that binding.

5.2.2 Numeric values

As in most programming languages, a FHiCL number can have up to four parts:

- The *sign* part consists of a single **+** or **−** character,
- The *whole* part consists of a non-empty sequence of digits, such as **0** or **123**; any extraneous leading zeroes will be ignored by the binding.
- The *fraction* part consists of a single **.** character followed by a possibly empty sequence of digits; any extraneous trailing zeroes will be ignored by the binding.
- The *exponent* part consists of a single **e** or **E** character, optionally followed by a sign, followed by a non-empty sequence of digits. Examples include **E5** and **e-23**.

All parts are optional with the restriction that at least one digit is specified in either the whole or the fraction part.

In addition to the above, FHiCL treats the literal **infinity** as a number. A sign may optionally precede this literal.

Supported examples include:

```
i : 14
t : .68
pi: 3.1415926
x : 1.23e2
y : -0.45600E+3
z : -infinity
```

A binding must take into account the mathematical value being represented as well as any constraints imposed by the underlying programming language.

5.2.3 Complex values

A FHiCL complex value is written as two numbers separated by a comma and surrounded by parentheses:

```
c1: (1, 2)
c2: (1.23, -3.1415926)
```

Whitespace before and after each number is optional.

5.2.4 String values

A FHiCL string is written as a sequence of characters usually enclosed within matching quotation marks. The quotation marks may be omitted, but only if the string contains no whitespace, punctuation, or other special characters:

```
s1: a
s2: ab
s3: string
s4: "string"
s5: 'string'
s6: "123abc"
s7: '123abc'
```

If the string is double-quoted, *escaped characters* will be interpreted as follows: **\n** as a newline character, **\t** as a tab, **\'** as an apostrophe, **\"** as a double-quote, and **** as a (single) backslash.[12] If the string is single-quoted, all characters are taken verbatim; escaped characters have no special meaning.

5.2.5 `@nil` value

The literal **@nil** serves as a placeholder value, distinct from all other FHiCL values. It is suitable for constructing a name-value pair when no other FHiCL value will do.

```
1 a: @nil
```

5.3 Sequence values

A FHiCL sequence starts with a left bracket and ends with a right bracket. These brackets surround a comma-separated list consisting of an arbitrary number of FHiCL values. Whitespace before and after each value is optional; thus, the following three sequences are considered identical.

```
1  q1: [ 1, 2, 3, 4 ]
2  q2: [ 1, 2
3      , 3, 4
4      ]
5  q3: [
6     1,
7     2,
8     3,
9     4
10 ]
```

Note that FHiCL sequences may be *heterogeneous*; that is, the *elements* (values in a sequence) may be classified differently from each other. For example, some may be numbers while others are not:

```
1 q4: [ 1, (2, 3.14), "a b", @nil, true ]  # 5 elements
2 q5: [ ]                                   # 0 elements (empty)
3 q6: [ [12, 34], 5 ]                       # 2 elements
```

However, any given binding can support such heterogeneity only to the extent that the underlying programming language supports it.[13]

A zero-based *subscript* (also known as an *index*) notation can be used to override an individual element, or even to extend a sequence with an additional element[14]:

[12] The binding will produce a diagnostic error message for any other escaped characters.

[13] This is rarely a restriction because, in practice, sequences tend overwhelmingly to be *homogeneous*.

[14] FHiCL sequences are *dense*: if a sequence contains n elements, their respective subscripts are always $0, 1, \ldots, n-1$. Extending a sequence will implicitly insert **@nil** values, if needed, to preserve this property.

```
fib   : [ @nil, 1, 1, "", 3, 5 ]  # 6 elements; heterogeneous
fib[0]: 0                         # @nil changed to '0'
fib[3]: 2                         # now a homogeneous sequence
fib[6]: 8                         # now 7 elements
fib[8]: 21                        # now 9 elements (fib[7] is @nil)
```

5.4 Table values

A FHiCL table starts with a left brace and ends with a right brace. Much like a document, the body of a FHiCL table consists of name-value pairs. Unlike a document, a FHiCL table cannot have a prolog (see section 8). The following document, for example, consists of a single name-value pair whose value is a table consisting of three pairs:

```
t: {
   a: 5
   b: 6
   c: { e: 2.718 }
}
```

The *member* notation can be used to override one of a table's values, or even to *inject* an additional pair into a table:

```
t.b: hi    # b's associated value is overridden with "hi"
t.d: 3.14  # table now gains a fourth pair
```

Note, however, that to override or inject any table values in this way, the full qualification of the relevant name must be specified. For example, in the following document

```
t2: {
   c: { e: 2.718 }
   c.e : "energy"     # error
}
```

the name **e** is only partially qualified when an attempt is made to override its value. The correct override syntax is to start at the outer-most name and to use the member and subscript notation as necessary:

```
t2: { c: { e: 2.718 } }
t2.c.e : "energy"     # OK
```

6 Names vs. keys

It is helpful to distinguish between a *name* and a *key*. In contrast to a name, a key can include member or subcript notation. Consider the following document:

```
t: {
   s: [ {entry  : 1},
        {another: 2} ]
   a: true
}
```

Table 1: List of names for the above FHiCL document.

Name	Value type
t	table
s	sequence
entry	atom
another	atom
a	atom

Table 2: List of keys associated with above document. All keys that begin with the name **t** are fully qualified keys.

Key	Value type
t	table
t.s	sequence
t.s[0]	table
t.s[0].entry	atom
t.s[1]	table
t.s[1].another	atom
t.a	atom
s	sequence
s[0]	table
s[0].entry	atom
s[1]	table
s[1].another	atom
entry	atom
another	atom
a	atom

The list of names for this document is shown in Table 1. Table 2 shows the complete list of keys associated with the same document. Note that each of the names listed in Table 1 is also a key listed in Table 2–i.e. names are a subset of keys. This distinction is important when considering references (see section 7).

7 References

Within some documents, a common scenario is to require that two (or more) name-value pairs with distinct names nonetheless provide the same value. The most obvious approach is simply to duplicate the value in each pair:

```
1 m: 1
2 n: 1
```

However, as is true when writing programs, such duplication becomes problematic over time because it is not obvious by inspection that these two values are intended always to be identical. As a result, if a future update were to change the value associated with **m**, the required corresponding change to **n** could be easily overlooked, especially if there were a great many intervening lines.

7.1 Substitutions

7.1.1 `@local::`

To help avoid such an unhappy scenario, FHiCL allows a value to refer to a previously-provided value:

```
1 m: 1
2 n: @local::m
```

A construction such as **`@local::m`** is known as a FHiCL *reference*. Each reference consists of the prefix **`@local::`** followed by a name (here, **m**) from an earlier name-value pair.[15] The following two documents are thus equivalent:

Document 1

```
1  a : false
2  s : [ a, b, c ]
3  t : { d: e }
4
5  a1: @local::a
6  a2: @local::s[1]
7  a3: @local::t.d
8  s1: @local::s
9  s2: [ @local::s, d ]
10 t1: @local::t
```

Document 2

```
1  a : false
2  s : [ a, b, c ]
3  t : { d: e }
4
5  a1: false
6  a2: b
7  a3: e
8  s1: [ a, b, c ]
9  s2: [ [a, b, c], d ]
10 t1: { d: e }
```

Notice that the subscript and member notations can be used for the **`@local::`** keyword in resolving references. This is true, in general, of any of the FHiCL references (see section 7.3).

When a binding processes such a reference, the name is looked up among the name-value pairs processed so far, and the corresponding value *substituted* (used in place of the reference). The value used for the substitution is the value associated with the name at the time the reference is parsed. As a result, future revisions of the configuration file which contain modifications to the value of the first will automatically be propagated to the second, and the two pairs will remain in sync.

In a single document, if the referenced name is given a new value, that new value will be used for subsequent references; previously processed references will not assume the new value.

[15] The binding will report an error during processing if the name has not yet been seen.

7.2 Splicing facilities

7.2.1 `@table::`

The **`@table::`** keyword is used to allow the contents of a referenced table to be spliced into the table in which it is invoked. For example, the following document:

```
t1: {
   a: 1
   b: [2,3]
}
t2: {
   @table::t1
   c: 4
}
```

produces the same parameter set as:

```
t1: {
   a: 1
   b: [2,3]
}
t2: {
   a: 1       # contents from
   b: [2,3]   # 'table1'
   c: 4
}
```

7.2.2 `@sequence::`

Similar to the **`@table::`** keyword, **`@sequence::`** is invoked to splice sequence contents into already-existing sequences. This document:

```
s1: [1,2,3]
s2: [ @sequence::s1, 4,5,6]
```

is equivalent to this one:

```
s1: [1,2,3]
s2: [1,2,3, 4,5,6]
```

7.3 References and fully qualified keys

The above **`@local::`**, **`@table::`**, and **`@sequence::`** FHiCL-reference keywords can be used only on fully qualified keys. Consider the following document:

```
t1: {
   t2: {
      test: 4
      list: [6,5,4]
   }
```

```
}
```

To access any of the above values using either substitution or splicing, the sequence of characters that follows `::` must be a fully qualified key. The following invocations represents valid FHiCL syntax (assuming the **`tab1`** definition is visible):

```
t2: {
   @table::t1.t2
   list: [ @sequence::t1.t2.list, 3,2,1 ]
}
a1: @local::t1.t2.list[0]
```

The above document is equivalent to:

```
t2: {
   test: 4
   list: [6,5,4, 3,2,1]
}
a1: 6
```

Note that a fully qualified key cannot be used until the definition of the outer-most name is complete. This means that this document:

```
t1: {
   m1 : { setting: 1 }
   m2 : { setting: @local::t1.m1.setting }  # error
}
```

is not a valid FHiCL document because a reference to a **`t1`** member is invoked before the closing brace of **`t1`** has been reached. A solution to this is to declare a name-value pair outside of **`t1`** that can be used inside of it:

```
global_setting: 1
t1: {
   m1: { setting: @local::global_setting }
   m2: { setting: @local::global_setting }
}
```

The disadvantage in this case, however, is that an extra name (**`global_setting`**) has been introduced at outer-most scope to support referencing within a local scope. Since this name is meant to merely support a single point of maintenance, and it is not meaningful to what the document is trying to represent, its inclusion could be unwanted. In addition, if many such names are introduced, the resulting parameter set can be unnecessarily large, and disentangling the meaningful name-value pairs from those that

are not meaningful can be difficult. This difficulty is resolved by the concept of prologs (see section 8).

8 Prologs

The purpose of a FHiCL *prolog* (also known as a *prolog section*) is to provide name-value pairs that can be referenced later in the document without appearing in the final parameter set.

A common use for a prolog is to provide alternative values from which to choose. The following document uses a prolog in this fashion such that only the 3-element sequence will appear in the parameter set:

```
1 BEGIN_PROLOG
2   opt1: [0, 1, 2]
3   opt2: [10, 11, 12, 13]
4 END_PROLOG
5 param: @local::opt1
```

The parameter set produced from the above document is indistinguishable from that produced from:

```
1 param: [0, 1, 2]
```

A document may contain as many or as few prolog sections as desired, so long as each starts with **BEGIN_PROLOG** and ends with **END_PROLOG**. No prolog may encompass another prolog; if there is more than one, they must appear strictly sequentially. Only comments may precede a prolog section; no prolog sections are permitted after a non-prolog name-value pair has appeared in the document.

Name-value pairs defined in a prolog can be overridden from outside of the prolog. Consider the following document:

```
1 BEGIN_PROLOG
2 a: { b: { c: 37 } }
3 END_PROLOG
4 a: { x: 12 }
5 b: @local::a
```

The value associated with **a** in the prolog is a table with an additional table nested inside of it. By reassigning the value of **a** outside of the prolog, the original prolog definition is erased, and the parameter set generated from the above document is equivalent to:

```
1 a: { x: 12 }
2 b: { x: 12 }
```

9 Includes

To deal with complexity, it may be desirable to assemble a larger FHiCL document from several smaller parts, with each fragment contained in its own file. Such assembly is made possible via FHiCL's **`#include`** directive.

The syntax for the directive is very strict in order to avoid possible confusion with a **`#`**-introduced comment: each **`#include`** is on a line by itself, with the **`#`** in the first column. There must be exactly one space following **`#include`**, and then a double-quoted string identifying the file name[16] of the target FHiCL document fragment.

```
#include "filename1.fcl"
#include "/path/to/filename2.fcl"
```

A document may be composed of as many such directives as desired. The binding will replace each directive with the document fragment contained in the corresponding named file.[17] A fragment may itself contain **`#include`** directives.[18] Although such directives are most commonly found at the start of a document, they may appear wherever a user finds convenient. However, it is ***strongly*** recommended that only prologs be placed in files that are **`#include`**d. This ensures that users can most easily glean the structure of a given FHiCL document under consideration.

10 Additional facilities

10.1 `@erase`

Specifying the **`@erase`** symbol as a value removes the previously defined name-value pair from the parameter set. For example, in this document:

```
a : {
   b1 : "some string"
}
a.b1 : @erase
```

the name **`b1`** is removed, and the table **`a`** is empty.

10.2 Modified binding operators and protection

In addition to the standard binding operator (`:`), there are two additional binding operators—**`@protect_ignore:`** and **`@protect_error:`**. These bindings are single symbols in that no space is permitted between the initial `@` and the word, or between the word

[16] It has become conventional to identify such files with the suffix **`.fcl`**.

[17] If it is set, the environment variable **`FHICL_FILE_PATH`** is consulted by the binding to locate a file so named. The value of this variable is the usual colon-separated paths typified by the bash standard **`PATH`** variable.

[18] However, no fragment may include itself, even indirectly.

and the trailing `:`. The three binding operators correspond to the following protection levels, ordered by increasing priority level:

None: A value bound to a name using `:` can be subsequently overridden. This protection represents the lowest priority level of the three.

Ignore: For a value bound using **`@protect_ignore:`**, subsequent assignments to the specified name are ignored.

Error: For a value bound using **`@protect_error:`**, a subsequent assignment attempt to the specified name is an error, for which a diagnostic message is provided by the binding language.

10.2.1 Protection inheritance

During assignment, a protection level is inherited from an enclosing name if the nested name has no specified protection. It is an error, however, if the enclosed name has a specified protection level of *ignore* when the enclosing name has a protection level of *error*. For example, the following FHiCL document:

```
1 a @protect_ignore: { b: 13 }
```

is equivalent to:

```
1 a @protect_ignore: {
2    b @protect_ignore: 13
3 }
```

This document, however,

```
1 a @protect_error: { b @protect_ignore: 13 }
```

is an error.

10.2.2 Protection when using `@erase`

Use of **`@erase`** on a name at a higher nesting level than that of a protected name ignores the protection of the item. For example:

```
1 a1: { b: { x @protect_ignore: 7 } }
2 a2 @protect_ignore: { b: { x: 7 } }
3 a1: @erase
4 a2: @erase
```

is equivalent to

```
1 a1: {}
2 a2 @protect_ignore: { b: { x: 7 } }
```

where the contents of **`a1`** have been erased, and those of **`a2`** have been retained.

10.2.3 Additional restrictions

It is an error to use a modified binding operator for an assignment to a name that already has a value:

```
a: 2
a @protect_ignore: 3 # error
```

Similarly, a local or fully qualified override shall honor protection, whereas a nested replacement shall not. For example:

```
a    : { b : { c @protect_error: 37 } }
d    : @local::a

a.b.c: 37               # error - protection honored
a    : 12               # OK    - protection overridden
d    : { b: { c: 43 } } # OK    - protection overridden
```

In the case of a local override attempt, the protection is respected:

```
a: {
   b: {
      c @protect_error: 37
      d: 31
      c: 43  #error - local override attempt
   }
}
```