CELCIO(1) **CELCIO(1)**

NAME

celcio – Celera C-language I/O subsystem generator.

SYNOPSIS

celcio [-r[123]] [-v] <datafile>

DESCRIPTION

Celcio reads a specification of all messages to be read and written in a contemplated production pipeline. Given the specification, *celcio* produces a C-language ".h" header file for the messages and a ".c" library file of routines that read and write the specified messages in both ASCII and binary formats.

Apart from the obvious productivity gained by not having to write I/O routines, the purpose of *celcio* is to permit one to import and export ASCII-encoded descriptions of C-structures in a consistent manner and to permit the binary exchange of such messages between components of an operating pipeline in an architecture-independent manner. *Celcio* is limited to non-circular structures that do not involve the data-overlays implied by unions. In a subsequent development we may build a system capable of automatically reading and writing a program's start and final state. *Celcio* would provide a substrate for such a system.

0. Syntactic Conventions:

We will be defining the syntax of a *celcio* specification file with a context free grammar. Non-terminals are enclosed in angle braces, e.g. `<spec>`, terminal string in quotes, e.g. `"array"`, and a rule consists of a nonterminal, an arrow (←), and a regular expression of terminals and nonterminals. For regular expressions we use the *egrep* operator symbols: | for alternation, juxtaposition for concatenation, * for 0 or more, + for 1 or more, and ? for 0 or 1. A terminal string in quotes may actually be an *egrep* regular expression in which case the terminal can be any string matching the expression.

White space may occur between any parts of speech separated by white space in the grammar specification, and indeed in some cases it is necessary to do so in order for *celcio* to be able to unambiguously parse the specification. Note carefully, however, that white space is not permitted within terminals. For example, in the left hand side – `"array" "[" <id> "]" "of" "float(32|64)"` – white space is permitted between "array" and "[", not permitted between "float" and "32", and required between "of" and "float".

In an attempt to convey some (but not all) type information within the grammar we often follow the name of a nonterminal with either a colon, ":", or an equal-sign, "=", followed by a type name chosen to suggest the meaning or type of the nonterminal. The use of an equal-sign implies that the type is assigned by the current context and a colon signifies the nonterminal should be of the given type. For example, we will class identifiers as being in one of the three categories – TYPE, FIELD, and ENUM – for datatype names, field names, and enum value names respectively.

1. Specification of Messages:

A *celcio* specification consists of an initial header section describing the manner by which message types will be enumerated followed by a series of data type declarations in a syntax that is a hybrid of the C and Pascal language styles.

```
<spec> ← <header> <declaration> *
```

Comments within a specification begin with "//" and continue to the end of the current line. They may begin where ever white space is permitted. In order to avoid naming conflicts with names that *celcio* must create in its translation of a specification, a user should not use identifiers that begin with the prefix "_CIO_".

In the header declaration, the user specifies the type name for an enumerative type that will contain an enum value for every distinct message declared in the remainder of the specification. Every *celcio* message is a struct whose first field will be a value of this type. The name of the field is specified followed by a template for constructing the names of the enum values. The template is a C-identifier that has an ampersand ('&') somewhere within it. Each *celcio* message structure has one or more unique 3-letter codes associated with it and it is these 3-codes that will be substituted for the template's & to construct the enum values.

```
<header> ←  "init" <id=TYPE> "=" <id=FIELD> ":" <template>

   <id>        ←  "[a-zA-Z_][a-zA-Z_0-9]*"

   <template>  ←  "([a-zA-Z_][a-zA-Z_0-9]*)?&[a-zA-Z_0-9]*"
```

For example, suppose one specifies the declaration "`init MessageTypes = mtype:AS_&_MESG`", and further suppose that in the remainder of the specification, messages with 3-codes `FRG`, `IFG`, and `SFG` are introduced. Then *celcio* will produce a declaration for `MessageTypes` that has the type "`enum { AS_FRG_MESG, AS_IFG_MESG, AS_SFG_MESG}`" and will explicitly place a field named `mtype` over this domain at the start of every message struct.

Each declaration associates a data type with an identifier. There are two types of declarations. Those that begin with the keyword "message" indicate to *celcio* that it should construct a read and write routine for this datatype. For such messages the data type must further be a structure. The remaining declarations, beginning with the keyword "type", are used to hierarchically build the datatype of a complex message. *Celcio* restricts struct definitions to the top level of a type declaration so that every struct has a type name associated with it.

```
<declaration>  ←  "type" <id=TYPE> "=" <data_type> | <struct_type=TYPE>
               |  "message" <id=TYPE> "=" <struct_type=MESG>
```

Datatypes can be built up with structure, list, sequence, and array constructors over a set of base types consisting of integers, reals, strings, enums, and chars.

- The sizes of the integer and float base types is indicated by following the keyword by the number of bits in the type. Integers can further be specified as unsigned with the "uint" keyword.
- A string is a sequentially allocated array of chars ends at the first 0-value element. This is C's concept of a string.
- An enum type is exactly as for C, the additional "one-code" enclosed in parens following each enum value is for the purposes of ASCII-based I/O (see Section 5) and is any printable, non-space character. The one-codes used within an enum must be distinct, whereas the enum values must be distinct across all enums, as dictated by C's semantics.
- A list type is a linked list of the indicated data type. *Celcio* provides primitives to walk and manipulate this datatype (see Section 4.).
- A sequence type is a sequentially allocated sequence of the indicated datatype where the length is *implicitly* part of the datatype. Its length need not be known before the object is read from input although an upperbound on it may optionally be given in the input as an aid to *celcio*'s memory allocator (see Section 5.).
- An array type is a sequentially allocated sequence of the indicated datatype where the number of elements is *explicitly* given by the identifier between the square braces. This identifier must be a field of type integer, list, sequence, or string in the current or an enclosing structure. If there are several nested structures that contain the array and have the given identifier as a field name, then the array identifier refers to the field declaration of the innermost enclosing structure. This field must be declared in such a

position that it is always read in before the array. If the type is list, sequence, or string, then the reference is implicitly to the length of the list, sequence, or string.

- A reference to a previous declared type identifier is a datatype.

```
<data_type>  ←  <int_type>
             |  "float(32|64)"
             |  "string"
             |  "enum" "{" ( <enum_con> "," )* <enum_con> "}"
             |  "list" "of" <data_type>
             |  "sequence" "of" <data_type>
             |  "array" "[" <id:FIELD> "]" "of" <data_type>
             |  <id:TYPE>

    <int_type>  ←  "u?int(8|16|32|64)"  |  "char"
    <enum_con>  ←  <id=ENUM> "("<1_code>")"
        <1_code>    ←  "[!-~]"
```

Lists, sequences, and arrays present three different ways to organize and model a homogenous collection of data elements. From a data structure point of view lists are realized as singly linked lists that must be accessed serially whereas arrays and sequences are realized as contiguous memory blocks that may be accessed by indexing. From an input/output perspective, the lengths of lists and sequences need not be known ahead of time whereas an array's length field must be input before the array. This makes space allocation for sequences difficult to realize with the same efficiency and overhead when a bound on its length is not given as part of the input. Thus one has a tradeoff: lists and arrays can be efficiently allocated, arrays and sequences can be efficiently accessed, and the length of lists and sequences need not be known in advance. One should select a representation with these tradeoffs in mind. One should further note that strings are identical to sequences in properties, save the manner in which the length of the string is encoded.

In its basic form, a structure declaration consist of the keyword "struct" followed by a list of field declarations inside a pair of curly braces. Each field declaration consists of a field name and a data type separated by a colon. Structures that have been declared to be messages must have one or more 3-codes associated with them to permit the *celcio* ASCII reader to determine the type of struct it is about to read. A 3-code is a C-identifier of length exactly 3. These are the 3-codes referred to earlier in describing the "init" line of a specification. How these are used in an ASCII encoding is described in the last section of this document.

```
<struct_type:TYPE>  ←  "struct" "{" <field_dec> + "}"

<struct_type:MESG>  ←  "struct" "(" <3_code=MESG> ("," <3_code=MESG>)* ")"
                              "{" <field_dec>+ "}"

    <field_dec>  ←  <id=FIELD> ":" <data_type>

        <3_code>  ←  "[_A-Za-z][_A-Za-z0-9][ _A-Za-z0-9]"
```

As an object flows through a pipeline, it is often the case that various stages add information to the object while some information becomes irrelevant after a certain point. To support this idea, *celcio* messages can be thought of as *containers* that can hold all the information that would ever be relevant to that type of object as it flows through the system. Enum-valued fields within the container are responsible for describing which fields/components of the message are currently meaningful. This could be accomplished, without further addition, by simply passing every field of the container between every stage of a pipeline regardless of its current utility and insuring that every unused field has been set to some legitimate but otherwise arbitrary value. *Celcio* provides additional constructs to avoid the overhead of having to read and write unused fields and to further explicitly document the usage of fields in a container message.

Within the field list of a *celcio* struct declaration, one may introduce if-else constructs where the conditionals are simple predicates based on enum-fields within a message. An if-else construct conditionally determines which of the two field sub-lists following the if-predicate and the else keyword are to be read and written by *celcio*. The constructs may be nested and chained. The predicates consist simply of a reference to a field of type enum that was read/written earlier in the processing of the message, followed by a contained-in ("<") or not-contained-in ("!<") predicate followed by a list of relevant scalar values in square brackets.

```
<field_dec> ←    "if" "(" <condition> ")" <field_list>
                 ( "else" <field_list> ) ?

    <field_list>   ←  <field_dec>
                   |   "{" <field_dec> +  "}"

    <condition>  ←  <id:FIELD> "!?<" "[" ( <id:ENUM> "," ) * <id:ENUM> "]"
```

Note that a given message container struct can have several 3-codes associated with it. The idea is that each 3-code determines a different usage pattern for the fields of the container. The implicit type field of the message is set to a different enum value depending on the 3-code, and conditional clauses may be formulated around the type field value to control field usage. Indeed, in an if-predicate, when the field reference is to the message's implicit type field, then in the list of scalar values one may use the 3-codes for the messages as opposed to the scalar value generated by the template given in the "init" declaration.

```
<condition>  <-  <id:FIELD> "!?<" "[" ( <3_code:MESG> "," ) * <3_code:MESG> "]"
```

We conclude this section with a complete example of a *celcio* specification. There are three types of messages all fitting in a container struct of type Fragment. The first field of this container contains the implicitly declared field "mtype" as described earlier. The field "iaccess" is not contained in FRG-messages. If the field "action" has value "AS_ADD" in a given message, then the message contains the "sequence" and "quality" fields, and the "screen" field if it is further an SFG-message. Note that the quality array is declared to be of the same length as the sequence.

```
init MessageTypes = mtype:AS_&_MESG

type ActionType = enum { AS_ADD(A), AS_DELETE(D) }

type ScreenMatch = struct {
                      what: int32
                      beg : int32
                      end : int32
                   }

message Fragment = struct(FRG,IFG,SFG) {
                       action    : ActionType
                       accession: int64
                       if (mtype !< [FRG])
                         iaccess: int32
                       if (action < [AS_ADD])
                         { sequence: string
                           quality : array [sequence] of uint8
                           if (mtype < [SFG])
                               screen: list of ScreenMatch
                         }
                    }
```

Figure 1: A complete *celcio* specification.

2. Semantic Checking and Field Reference Binding Details:

To the extent possible, *celcio* semantically checks its input as it parses it in a forward scan through the specification. During the scan the following conditions are checked in addition to the restrictions imposed by the syntax of the previous section. Violation of any of these conditions results in an error message and compilation terminates prematurely.

- All usages of a type or enum-value identifier occur after their defining usage. This is always possible to arrange in *celcio* as it does not permit circular data structures. Note that the references to field identifiers as an array's index or part of a container conditional are exempted. Their binding to a defined instance is deferred to the second semantic pass.
- Duplicate definitions of an enum value or a datatype name are detected, as are duplicate definitions of a field name within a struct (including the implicitly defined message type field).
- Duplicate definitions of 3-codes for a struct, or for fields within a struct are detected. Also duplicate 1-codes within an enum declaration are detected.

We term the usage of a field name as an array index or in a container conditional, as a field reference. The binding of a field reference to a field definition takes place in a second semantic pass. This is necessary, because it is often impossible to *simultaneously* arrange for type references and field references to always follow their defining instances. Consider the following example:

```
type A = struct { … array [x] of int32 … }
message B = struct { … x: int16; … y: list of A; … }
message C = struct { … x: int32; … z: A; … }
```

Assume the structure declaration for A does not contain a field named x, thus the reference to x as an array index must be resolved to a containing object. Indeed in B-messages the field reference is bound to a field of type int16 and in C-messages it is bound to a field of type int32. This late binding of field references permits one to declare datatypes that may be differentially instantiated depending on the context in which they are being used.

In the second semantic sweep, all implied bindings of a field reference are examined to insure that in each context:

- The reference does have some field instance to bind to.
- The type of the bound field is correct.
- The enum values in a conditional are part of the bound field's enum range.

Failure of any of these checks results in an error and premature termination of compilation. In addition a number of warnings, that do not prevent compilation, but are indicative of potential problems, are also detected and reported to the command line port as follows:

- Any type definition not used in a message or a subtype of a message is flagged.
- In the expansion of the definition of a message, *celcio* symbolically evaluates the conditionals within it and detects the following:
  1. Conditionals that have no effect, i.e. are equivalent to "true".
  2. Conditionals that are equivalent to "false", implying the field sublist within it is never part of a message.
  3. Field references that occur in conditional contexts that do not necessarily guarantee that the field to which the reference is bound is a part of the message.

We give an example of a specification invoking the last three warning conditions as an aid to their comprehension.

```
message bad = struct(BAD) {
        fd1(fd1): enum { A(A), B(B), C(C) }
        if (fd1 !< [C])
          { fd2(fd2): int32
            if (fd1 < [A,B])             ← Always true
              fd3(fd3): int32
          }
        else if (fd1 !< [C])             ← Always false
          fd4(fd4): int 32
        fd5(fd5): array [fd2] of char    ← fd2 not defined when fd1 = C
      }
```

3. Translation to Data Declarations:

In response to a specification, say "`fragment`", *celcio* will produce two files – "`fragment.h`" and "`fragment.c`", where the file with the ".h" extension contains a C type declaration for every type specified in the input. Here we detail the exact translation between the input *celcio* specifications and the output C-declarations, including an example to aid one's intuition.

Before outputting any user declared types, *celcio* outputs typedefs for the base types `int8` through `int64`, `uint8` throught `uint64`, `float32` and `float64`, tailored to the particular machine on which *celcio* is being run. For example, "`typedef long long int64;`" on 32-bit machines and "`typedef long int64;`" on 64-bit machines. After outputting declarations for the user declared types, *celcio* concludes its header file with the declarations for the I/O routines that it will compile and make externally visible in its ".c" extension file.

Consider the sample specification for "Fragment" messages given in Figure 1 earlier. The datatypes in this *celcio* specification would result in the C-declarations shown in Figure 2. The first type definition is for the enum that captures the three kinds of messages specified by the user. The type name for this enum is as specified in the `init`-line, and the enum values consist of the template with its "&" replaced by the 3-codes of each message. The enum values are mapped to consecutive integers starting at 0. The type definition for "ActionType" is straightforward, but note that each enum value is mapped to the value of the ASCII code for its associated 1-code. The other key thing to notice is that the struct definition for "Fragment" messages has a field "mtype" of type"MessageTypes" as its first field as dictated by the `init`-line of the specification. Note that strings (and also arrays) are mapped to pointers to the base type, and that lists are always of type "`void *`" (more on this later).

```
typedef enum { AS_FRG_MESG = 0,
               AS_IFG_MESG = 1,
               AS_SFG_MESG = 2 } MessageTypes;

typedef enum { AS_ADD    = ‘A’,
               AS_DELETE = ‘D’ } ActionType;

typedef struct { int32 what;
                 int32 beg;
                 int32 end;  } ScreenMatch;

typedef struct { MessageTypes mtype;
                 ActionType   action;
                 int32        iaccess;
                 char *       sequence;
                 uint8 *      quality;
                 void *       screen;   } Fragment;
```

Figure 2: C-type declaration for the *celcio* specification of Figure 1.

To formally describe the translation to C-types, it helps to simplify the complete grammar of the previous section to the features essential to characterizing the translation. Specifically, we drop all type information, all mention of 3-codes (necessary only for specifying instances of objects), and simplify structures to being just a list of field declarations – the complication of conditional clauses is ignored. This results in the simple grammar below.

```
          <spec>  ←  <header> <declaration> *
        <header>  ←  “init” <id> “=” <id> “:” <temppart1>“&”<temppart2>
   <declaration>  ←  “type|message” <id> = (<data_type>|<struct_type>)
     <data_type>  ←  “float(32|64) | u?int(8|16|32|64) | char”
                  |  “string”
                  |  “list” “of” <data_type>
                  |  “sequence” “of” <data_type>
                  |  “array” “[” <id> “]” “of” <data_type>
                  |  “enum” “{” ( <enum_con> “,” )* <enum_con> “}”
                  |  <id>
      <enum_con>  ←  <id> “(”<1_code>“)”
   <struct_type>  ←  “struct” “{” <field_dec> + “}”
     <field_dec>  ←  <id> “:” <data_type>
```

To formally define the translation we define the function T that maps a parse tree of a celcio specification to the string containing the desired C type definitions. We define T by recursively defining its action on each production of the simplified grammar above. The result in each case is a string where terminal letters are bold and T is recursively applied to non-terminals on the r.h.s. of the production as needed. For example,

T(<A> ← <B> <C> “;”) ⇒ **some** T(<B>) **text**

In the cases where the yield of a non-terminal is needed in the functional result, we introduce a variable in the non-terminal's angle brackets and use it in the string, e.g. in

T(<A> ← <**x** ∈ id>) ⇒ **x**

the variable ‘x’ matches the yield of <id> in the particular specification.

We now define the action of T on each production of the grammar immediately above in the order they are listed followed immediately by a textual explanation.

T(<spec> ← <header> <declaration> *) ⇒ T(<header>) (T<declaration>)*

The translation of a specification is the translation of its header followed by the translation of each of its declarations in the order given.

T(<header> ← “init” <**y** ∈ id> “=” <id> “:” <**u** ∈ temppart1>“&”<**v** ∈ temppart2>)

⇒ **typedef enum { uC_1v = 0, uC_2v = 1, … uC_kv = k-1, } y;**

The translation of the specification header is a C-type definition for type name “y” that is an enum type whose constants are the template of the header with the ampersand replaced by the 3-codes, “C_1, C_2, … C_k”, of every message occuring in the declarations that follow. The values of the constants are set to 0 through k-1.

T(<declaration> ← “type” <**x** ∈ id> = (<data_type>|<struct_type>))

⇒ **typedef** T(<data_type>|struct_type>) **x;**

The translation of a declaration is a C-typedef for the id denoted by “x” where the type is given by the translation of the declaration's data type.

T(<data_type> ← “**x** ∈ float(32|64)|u?int(8|16|32|64)|char”) ⇒ **x**

The translation of a scalar data type other than an enum-type is the *celcio* string for the type denoted by “x” in the above equation.

T(<data_type> ← “string”) ⇒ **char ***

The translation of a string data type is the type “char *”.

T(<data_type> ← "list" "of" <data_type>) ⇒ **void ***

The translation of a list data type is the type "void *".

T(<data_type> ← "sequence" "of" <data_type>) ⇒ T(<data_type>) *****

The translation of a sequence data type is a pointer to the C-type that is the translation of the array's base data type.

T(<data_type> ← "array" "[" <id> "]" "of" <data_type>) ⇒ T(<data_type>) *****

The translation of an array data type is a pointer to the C-type that is the translation of the array's base data type.

T(<data_type> ← "enum" "{" (<enum_con> ",")* <enum_con> "}")

⇒ **enum {** T(<enum_con>) **,**)* T(<enum_con>) **}**

T(<enum_con> ← <**x** ∈ id> "("<**a** ∈ 1_code>")") ⇒ **x = 'a'**

The translation of an enum data type is a C-enum type where the enum constants are as in the *celcio* type, and where each constant is set to the value of its 1-code character.

T(<struct_type:MESG> ← "struct" "{" <field_dec> + "}")

⇒ **struct {** F(<header>) T(<field_dec>) + **}**

T(<struct_type:TYPE> ← "struct" "{" <field_dec> + "}")

⇒ **struct {** T(<field_dec>) + **}**

The translation of a struct data type is a C-struct where the field declarations are the translations of the fields of the *celcio* specification. In the case that the struct is a message, a field definition for the type of the message is inserted as the first field. This is captured formally as the F-translation of the header of the file.

F(<header> ← "init" <**y** ∈ id> "=" <**x** ∈ id> ":" <template>) ⇒ **y x;**

The F-translation of a specification header is a field declaration for field "x" of type "y".

T(<field_dec> ← <**x** ∈ id> : <data_type>) ⇒ T(<data_type>) **x;**

The translation of a field declaration for "x" is a C field-declaration for "x" where the type is the translation of the field's data type.

T(**<**data_type> ← <**x** ∈ id>) ⇒ **x**

The translation of type name is the type's name "x".

4. Translation to I/O Routines:

In response to a specification, say "`fragment`", *celcio* will produce two files – "`fragment.h`" and "`fragment.c`", where the file with the ".c" extension contains C code that will read and write every declared message in both ASCII and binary formats. The nature of this code is not of any concern to the *celcio* user other than that they understand that it must be compiled and linked in to any program that will use the defined data interface. The user should note carefully that two different .c files produced by *celcio* may not be used simultaneously – if a program reading and writing the union of the messages in the two *celcio* specifications is desired, then one should merge the two specification files prior to compilation into a single .h and .c file.

The externally visible routines in the ".c" file always present the same interface/abstraction to the user and are declared at the end of the ".h" file that should be included in the users program. This interface consists of 6 routines for reading and writing messages, 7 routines for walking and manipulating lists, and 2 concerning sequences.

The IO primitives center on use of "CelcioUnit"s that maintain the state of a series of message reads or writes to a given file or IO port. One can establish CelcioUnits for reading and writing, perform a series of reads or writes on a CelcioUnit correctly established for the given mode of operation, query the type of any message, and destroy a CelcioUnit when it is no longer of use. Each primitive is listed below with a description of its arguments and action.

- **`CelcioUnit *CreateCelcioReader(FILE *ifile, int buffer_size);`**

  Prepare to read input from the given "stdio.h" FILE "ifile". This file should already be opened for reading but may have its read cursor positioned anywhere within the file so long as a legitimate stream of celcio messages begins at the current position. The returned "CelcioUnit" object keeps track of the current state of the *celcio* read sequence that will ensue on the file. The parameter "buffer_size" gives the number of bytes to initially allocate for the storage of successive messages to be read via the input unit. If set to 0, *celcio* uses its default size.

- **`CelcioUnit *CreateCelcioWriter(FILE *ofile, int binary);`**

  Prepare to write input from the given "stdio.h" FILE "ofile". This file should already be opened for writing. The returned "CelcioUnit" object keeps track of the current state of the *celcio* write sequence that will ensue on the file. If the parameter "binary" is non-zero then celcio will write binary message, otherwise it will write ASCII messages.

- **`void *ReadCelcioMesg(CelcioUnit *input_unit);`**

  Read the next message from the stream coded in the created "CelcioUnit" input_unit. Return NULL if at the end of file, or a pointer to the next message if the read is successful. The memory in which the message is placed is owned by *celcio* and will be reclaimed when the next read request is made of the input unit.

- **`<Message Enum Type> TypeOfCelcioMesg(void *mesg);`**

  Return the enum value encoding the type of the celcio message pointed at by "mesg". The return type is as given in the "init" line of the defining specification. Once the type of a message is known one may caste it to the appropriate type for the message.

- **`void WriteCelcioMesg(CelcioUnit *output_unit, void *message);`**

  Write the *celcio* message pointed at by "message" to the file "ofile". This file is expected to be open for writing.

- **`void DestroyCelcioUnit(CelcioUnit *unit);`**

  Reclaim the memory associated with the *celcio* unit "unit" and release it. The "stdio.h" FILE associated with this port is not closed, the user is expected to do so.

Based, on the celcio specification of Figure 1, we give in Figure 3 a small example of a program that reads the messages on the file "SampleInput" and writes all those that are of type "FRG" onto the standard output. While the program is complete, it does not engage in the usual checks of call results in order to keep the example small.

```
#include <stdio.h>
#include "fragment.h"

int main()
{ FILE *file;
  CelcioUnit *input, *output;
  void *mesg;
  ifile  = fopen("SampleInput","r");
  input  = CreateCelcioReader(ifile,100000);
  output = CreateCelcioWriter(stdout,0);
  while ((mesg = ReadCelcioMesg(input)) != NULL)
    if (TypeOfCelcioMesg(mesg) == AS_FRG_MESG)
      WriteCelcioMesg(output,mesg);
  DestroyCelcioUnit(input);
  DestroyCelcioUnit(output);
  fclose(ifile);
  exit (0);
}
```

Figure 3: Sample usage of *celcio*'s IO routines.

A *celcio* list is mapped to an opaque pointer (i.e. void *) to insure that user's manipulate the data abstraction through the seven routines or methods below.

- **`CelcioWalker *StartCelcioWalk(void *list, CelcioUnit *unit);`**

  All traversals or modifications of a list begin by calling "StartCelcioWalk" on the list in question, or NULL if a new empty list is desired. The primitive returns a pointer to a CelcioWalker object that maintains the current state of the list traversal. The traversal cursor maintained by a CelcioWalker is to a position that is always between elements of the list. Initially, the cursor is before the rightmost/first element of the list. If "unit" is not NULL, then the space management for adds and deletes for this walk will take place in the memory space managed by the given unit. In this event it should be the case that the list is part of a message read by this unit or is NULL. If "unit" is NULL, then list elements are malloc'd and free'd in response to add and delete requests. A list may have only one active walker associated with it at any time.

- **`int   AdvanceCelcioWalk(CelcioWalker *walk, int right);`**

  Each call to "AdvanceCelcioWalk" with a given walker moves its cursor one position to the left or right depending on whether the parameter "right" is non-zero or not. If the cursor is already at the end of the list in the requested direction, then the call has no effect but returns a non-zero value, effectively providing the user with a test for whether they are at the left or right end of the list.

- **`void *GetCelcioWalkObject(CelcioWalker *walk, int right);`**

  A call to "GetCelcioWalkObject" returns a pointer to the object held by the element immediately to the right or left of the current position of "walk" depending on whether the parameter "right" is non-zero or not. It is the responsibility of the user to caste the opaque pointer returned to a pointer to an object of the appropriate type for the list being walked. If there is no object in the given direction from the cursor, then NULL is returned.

- **`int  AddToCelcioWalk(CelcioWalker *walk, void *obj, int size);`**

  Insert at the current cursor point a new element to the list where "obj" is a pointer to the base element value to be inserted and "size" is the size of the base element in bytes. A copy of the base element is inserted into the list, thus the base element passed as an argument does not need to persist after the call. The routine returns a non-zero value if it was unable to allocate space for the new element, and in this event has no effect on the list.

- **`int   DelFromCelcioWalk(CelcioWalker *walk, int right);`**

Delete the element immediately to the right or left of the cursor depending on whether “right” is non-zero or not, respectively. If the designated element does not exist then the routine has no effect save that it returns a non-zero value indicating this condition. One must be careful to free any memory referred to by the element being deleted in order to avoid a memory leak.

- **`void *EndCelcioWalk(CelcioWalker *walk);`**

  Terminate the traversal of the list being manipulated by the walker “walk”. The walker object is destroyed and its storage freed. A pointer to the list object is returned. If the list has been modified by additions or deletions, then it is imperative that any references to the list object be set to this new reference as the underlying list has changed.

- **`long  CelcioListLength(void *list);`**

  Return the length of the celcio list pointed at by “list”. Note that this primtive takes time proportional to the list.

We now give an extended example of the use of these primtives in Figure 4 using the “Fragment” message example of Figure 1. The example takes the form of a routine “WalkExamples” that is passed a Fragment structure as an argument. It is immediately checked if this is an “SFG” message and if not then the routine

```
void WalkExamples(Fragment *fmsg, CelcioUnit *input)
{ ScreenMatch *scm, sitem;
  CelcioWalk  *walk;
  int          i;

  if (TypeOfCelcioMesg(fmsg) != AS_SFG_MESG) return;

  printf("\nScreen Matches\n");
  walk = StartCelcioWalk(fmsg->screen,NULL);
  while ((scm = (ScreenMatch *) GetCelcioWalkObject(walk,1)) != NULL)
    { printf("  %d[%d,%d]\n",scm->what,scm->beg,scm->end);
      AdvanceCelcioWalk(walk,1);
    }
  EndCelcioWalk(walk);

  walk = StartCelcioWalk(fmsg->screen,input);
  for (i = 0; i < 3; i++)
    { sitem.what = 22;
      sitem.beg  = 50*i;
      sitem.end  = sitem.beg + 50;
      AddToCelcioWalk(walk,(void *) (&sitem),sizeof(ScreenMatch));
      AdvanceWalk(walk,1); /* if want Queue list build, else get Stack build */
    }
  fmsg->screen = EndCelcioWalk(walk);

  walk = StartCelcioWalk(fmsg->screen,input);
  for (i = 0; (scm = (ScreenMatch *) GetCelcioWalkObject(walk,1)) != NULL; i++)
    if (i%2)
      DelFromCelcioWalk(walk,1);
    else
      AdvanceCelcioWalk(walk,1);
  fmsg->screen = EndCelcioWalk(walk);
}
```

Figure 4: Sample usage of *celcio*’s list routines.

returns immediately. Otherwise the message has a "screen" list and the following manipulations occur. First the list is walked in order and the contents of each Screen Message record printed on a line of the standard output. In the next code block 3 ScreenMatch elements are inserted at the front of the list. Note carefully that if the call to "AdvanceWalk" is present then the result is "22[0,50] 22[50,100] 22[100,150] <original list>", whereas if it is left off then the list is "22[100,150] 22[50,100] 22[0,50] <original list>". In the final traversal, every other element of the list is deleted from the list.

One must be careful to appreciate the memory management issues surrounding the AddToCelcioWalk and DelFromCelcioWalk routines in order to avoid creating memory leaks in their application. Each input CelcioUnit maintains its own memory space in which it allocates the messages it returns. At other times a user may use C stdlib's malloc and free to build messages that they wish to output. Deleted cells must be reclaimed by the appropriate manager. Added cells can be allocated by either manager, but if malloc'd then these cells must later be free'd if a leak is to be avoided, and if allocated in the read managers space then these cells cannot later be free'd if a core dump is to be avoided. Generally we suggest two templates: (1) for a read message that is to be modified before being output, use the unit's manager, and otherwise (2) construct a message using malloc and free, including lists which are initially NULL and built up by adds. For template (2), constructed lists must be freed by deleting every element when it is time to free the overall message.

A *celcio* sequence is equivalent to a C-array for the object save that one gets the length of the sequence (tucked away in a long integer prefacing the array) by calling "CelcioSeqLength". The two other necessary primitives are "CreateCelcioSequence" and "DestroyCelcioSequence", needed when on is building a message from scratch in memory.

- **`long  CelcioSeqLength(void *seq);`**

  Return the length of the celcio sequence pointed at by "seq". Note that you must coerce the type of the sequence pointer to "void *" for the call.

- **`void *CreateCelcioSequence(long length, int size);`**

  Allocate a sequence object of "length" elements of "size" bytes each. If space is not availae, NULL is returned, otherwise a pointer to the array portion is returned, wherein one may then assign the elements.

- **`void DestroyCelcioSequence(void *seq);`**

  Free the memory of the sequence object pointed at by "seq". This object must either have been read by "ReadCelcioMesg" or created by "CreateCelcioSequence".

5. ASCII Format Conventions:

The design of the ASCII formating conventions for *celcio* data structures aims to be as flexible as possible while still permitting detection of errors as close to the source as possible. Basically, an input stream may be thought of as a sequence of tokens separated by white space, where a white space character is any of the symbols blank, horizontal and vertical tab, form-feed, carriage return, and new-line. Tokens may appear anywhere on a line and need not be separated by white space when the boundary between the tokens can be unambiguously identified (e.g., an integer followed by a left parenthesis, but not two consecutive integers).

There is a limit on line length, currently set at 2048. This is controlled by the defined constant **LINEMAX** in the file "celcio.h" for those that might need to enlarge it.

The comment convention for a *celcio* ASCII data file is that the remainder of a line from a #-token onwards is ignored. In addition, there is a special identifier token that is also always ignored, but the rest of the line following it is parsed. This token is an '@'-sign followed by any C-identifier and formally matches the regular expression "`@[_A-Za-z][_A-Za-z0-9]*`". These tokens are useful if one wishes to give the field names of a structure or the type name of an object within a datafile as a form of further information for a human reader. We will illustrate its use in some examples following the complete specification.

In order to rigorously define the syntax of celcio ASCIO I/O, we introduce the following inductive definition of a celcio datatype that distills the type information of a celcio specification to its barest essence.

<type> ← **struct {** <type$_1$>**;** <type$_2$>**;** … <type$_n$> **}**
| **struct (xxx) {** <type$_1$>**;** <type$_2$>**;** … <type$_n$> }
| **list of** <type>
| **sequence of** <type>
| **array** [n] **of** <type>
| **string**
| **integer**
| **float**
| **char**
| **enum (x$_1$,x$_2$, … x$_n$)**

Structures have been separated into those that are messages and thus have an associated 3-code "xxx", and those that are not. To simplify matters all integers types and all float types have been collapsed into a representative "integer" and "float" type, and the fact that the base type of strings is restricted is ignored. The type of a user-defined enum is captured in the definition by the list of 1-codes – x_1, x_2, … x_n – associated with each of its possible values.

We can now give the exact manner of encoding an instance of each type construct in the definition above followed by a brief text explaining it.

struct { <type$_1$>**;** ⇒ **{** <object$_1$> …<object$_n$> **}**
<type$_2$>**;**
…
<type$_n$>**;**
}

A struct object is encoded as a token consisting of an opening curly brace, followed by the encodings of the objects for each of its fields' values in sequence, and ending with a closing curly brace token.

struct (xxx) { <type$_1$>**;** ⇒ **{xxx** <object$_1$> …<object$_n$> **}**
<type$_2$>**;**
…
<type$_n$>**;**
}

A message struct object is encoded as a 4-letter token consisting of an opening curly brace and the message's 3-code, followed by the encodings of the objects for each of its fields' values in sequence, and ending with a closing curly brace token.

list of <type> ⇒ (<object$_1$> <object$_2$> …<object$_n$>)

A list object is encoded as a token consisting of an opening parenthesis, followed by a series of the encodings of the objects in the list, and ending with a closing parenthesis token.

sequence of <type> ⇒ `“\"[0-9]*”` <object$_1$> <object$_2$> …<object$_n$> "

A sequence object is encoded as a token consisting of a double quote optionally followed by an unsigned integer, followed by a series of the encodings of the objects in the sequence, and ending with double quote. The integer, if present, tells *celcio* how big a block to initially allocate for the sequence – this helps the memory manager accommodate the variable-length nature of a sequence. It is, of course, best if this integer is greater than or equal to the actual length.

array [n] **of** <type> ⇒ **[** <object$_1$> object$_2$> …<object$_n$> **]**

An array object is encoded as a token consisting of an opening square bracket, followed by a series of the encodings of the objects in the array, and ending with a closing square bracket token.

string ⇒ `“\"[0-9]*”` <char$_1$> <char$_2$> …<char$_n$> `"`

A string object is encoded as a token consisting of double quote optionally followed by an unsigned integer, followed by a series of the chars terminated by a double quote token. . The integer, if present, tells *celcio* how big a block to allocate for the string – the number of characters in the sequence must not be greater than this integer (but may be less than it).

integer ⇒ “C integer constant” (`“-?[0-9]+”`)

An integer object is encoded according to C conventions for integer constants.

float ⇒ “C floating point constant”
(`“([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][+-]?[0-9]+)?”`)

A floating point object is encoded according to C conventions for floating point constants.

char ⇒ `“([!-~]-[ @#(“\])|\\[\001-\177]|\\[0-7]* ”`

A character object is encoded as either (a) its ASCII character if it is a printable character other than blank, backslash, double quote, at-sign, pound-sign or left paren, (b) a backslash followed by an character (printable or not), or (c) a backslash and the ASCII octal encoding of the symbol.

enum **(x_1, x_2, …, x_n)** ⇒ `“[`$x_1x_2…x_n$`]”`

An enum object is encoded as the character corresponding to one of the 1-codes chosen to denote its particular range of values.

There is one variation on the rules above: when a message struct is being used as a subcomponent of another message and there is only one 3-code for the given message struct, then one may omit the 3-code that usually follows the open curly brace, as it can be inferred from context.

A small difficulty with ASCI format is that a slight bit of precision is lost on floating point values when they are written out, about ½ digit.

While sequence types can create some potentially inefficiencies with regards to memory management, this is only true if the input does not contain a size prefix that gives the length or a reasonable upper-bound on the length of the sequence. One should note carefully, that *celcio*'s writers always output the length of a sequence as part of its ASCII representation, therefore, the only time sequences should present any loss of efficiency is the first time they are consumed by a *celcio*-implemented IO utility.

We close with several examples of how the same Fragment message (our running example from Figure 1) might be encoded within the above framework. In the first case, the message is verbose in that every field name is commented and lots of white space is used. In the second case, comments are ommited and a terser layout is employed. Finally, in the last example, the instance is encoded with as few characters as possible, insert white space only where necessary.

```
{SFG                  # Encoding of a fragment record
@act A
@acc 1700006130
@iac 232
@seq “Crazy\ man”
@qlt [1 1 1 1 1 2 3 3 3]
@scn (
    { @what  1
      @beg   0
      @end   5
    }
    { @what  3
      @beg   6
      @end   9
    }
  )
}

{SFG A
 1700006130 232
 “Crazy\ man”
 [1 1 1 1 1 2 3 3 3]
 ( {1 0 5} {3 6 9} )
}

{SFGA1700006130 232”Crazy\ man”[1 1 1 1 1
2 3 3 3]({1 0 5}{3 6 9})}
```

AUTHORS

Gene Myers:

Created November 30, ‘99