recio: Record Stream I/O Library Version 1

CML00001-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

August 16, 2016

Contents

1 Introduction

The Code Magus Limited Record Stream I/O (`recio`) Library provides a generic interface for programs to access file systems and other data access mechanisms. This generalised interface, in turn, refers to the specialisations that deal with the various file systems and data access mechanisms to satisfy the requirements of various situations and environments. The using program is thus not directly bound to the constraints of these various situations and environments. The library provides a separate interface for the implementation of these specialisations whose details are supplied by a configuration file in each case.

The `recio` library refers to the specialisations as *Access Methods* and refers to the configuration file as an *Access Method Definition*. The access method definition describes the access method implementation module and location within the hosting system. Access methods take parameters which are referred to as options, and it is the access method definition that names, describes and states the constraints on the values of these options. The access method definition also explicitly states the methods that the access method implements.

Figure 1 on page 3 shows the structure of the relationship between the consumer and supplier components in the architecture of the `recio` record stream library. The figure illustrates the relationship amongst the components and also introduces some of the terminology used in this briefing.

2 Benefits of using `recio`

There are a number of benefits to this approach, some of which are the following:

- Programs that bind to the `recio` library for purposes of accessing file systems and external data sources, provided that their interpretation or preparation of the content (i.e. individual records) can be done taking appropriate meta-data mappings into account (for example, the mechanism provided by the Code Magus Limited Object Types Library), can be written in a platform independent manner. For example, the Code Magus Limited File Tools Copy program uses the `recio` library, and allows the tool to be run on z/OS, Unix/Linux systems, Windows and VOS without changes. Additionally, this program uses the Code Magus Object Types Library [5] and the Code Magus Expression evaluation library where the content is required to be inspected under specific meta-data bindings in order to evaluate queries over the content.
- Programs that bind to the `recio` library for purposes of accessing file systems and external data sources can be written in a manner which is independent of the details of those file systems and external data sources. For example, Eresia scripts

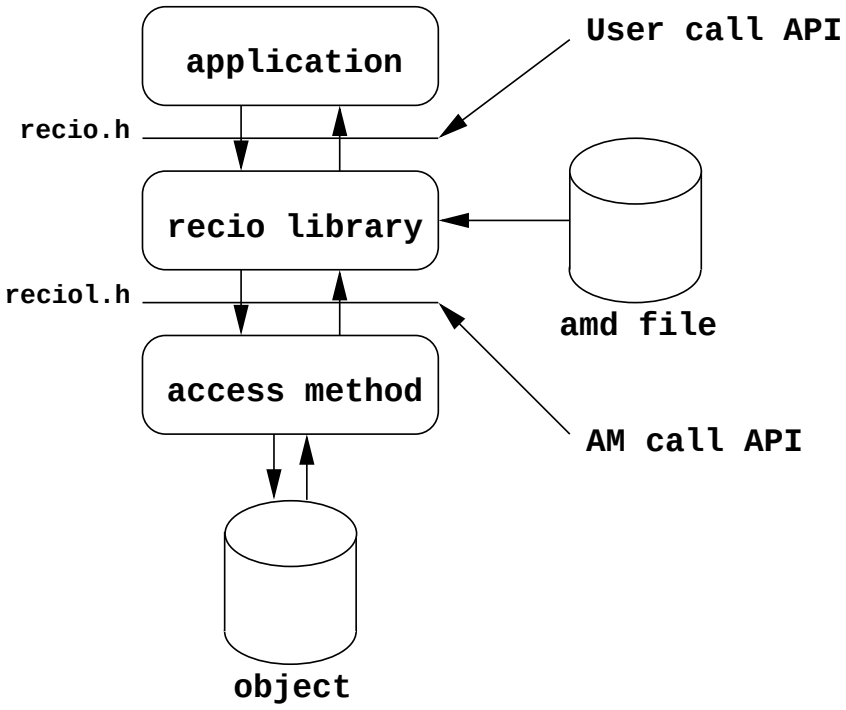


Figure 1: Overview of the `recio` record stream I/O Library and its relation ship to its supplier and consumer components

can access to the `recio` library through a Type A Interface [6]. This allows the scripts to process files that are foreign to the hosting platform by reading and writing data on local or remote platforms such as z/OS, Unix/Linux systems, Windows and VOS.

- Programs that bind to the `recio` library for purposes of accessing file systems and external data sources can gain access to proprietary data sources without embedding the details of that proprietary data source, provided that an appropriately developed access method is made available. For example, contents of DB2 image copy datasets can be interpreted as records or rows of file stream and supplied to the program as though the contents of a regular file without the program dealing with any of the internal specifics of the image copy structure.
- File systems and data sources that are natively hosted on one system can be made available to programs on other systems by using the included data server. The data server runs on various platforms and includes the validation of credentials against the access requirements of the objects being accessed. For example, RACF defined users on z/OS can gain access to their datasets from foreign platforms with the knowledge that their datasets are being controlled by the RACF defined profiles.
- Further abstraction from the actual object name can be achieved by using the

DATASET access method. This access method uses the object given to look up in a known or specified catalog for the actual open specification to use. This also allows the exact same open string on one system to resolve to a different data object than another system. For example an MVS test and production system implement different naming conventions for the same dataset content; this difference is hidden within the catalog entry on each system and the same open specification can be used on both platforms.

3 Access Methods

There are a number of access methods supplied by Code Magus; the most common being:

- TEXT [8]. Supports access to delimited record text files implemented on byte-stream file systems.
- BINARY [1]. Supports access to fixed and variable length record files implemented on byte-stream file systems.
- DATASET [2]. Supports abstraction of open specifications for any access method by implementing a catalog containing references to the actual open specification.
- IMAGE [9]. Supports access DB2 imagecopy files.
- MVS [4]. Supports access to MVS datasets using any other defined access method, but first allocating the file with MVS JCL syntax.
- DIR [3]. Supports directory access on Unix and Windows based systems and Catalog access on MVS.
- REMOTE [7]. Supports access to recio objects on remote hosts.
- SPOOL. Supports the ability for Eresia scripts to access the Eresia job spool via the Eresia local spool access method.

4 recio User Call Interface

User programs access the recio library through a call interface which creates *stream* structures on successful open (recio_open) operation and provides read and write (recio_read and recio_write) operations to move data into and out of the program. Also available are point and tell (recio_point and recio_tell) operations for navigating files. The stream is closed using the recio_close function. There are additional utility functions (consult the header file recio.h for further details).

The functions prototypes are made available to the application program by including the header file `recio.h`.

```
#ifndef RECIO_H
#define RECIO_H
  /* File: recio.h
   *
   * This header describes the interface to the Code Magus Limited Record I/O
   * Module. The Record I/O Module allows applications which expect a record
   * stream as input to be able to use user and third party defined record
   * streams for the I/O of their applications.
   *
   * The methods required to support the I/O processing of a particular stream
   * type are supplied by a loadable module for the stream type. The name and
   * location of this loadable module is supplied by a configuration of the
   * stream definition. The stream definition is located either in a standard
   * place for each platform or supplied by an environment variable value.
   *
   * This Library is not for distribution.
   *
   * Author: Stephen Donaldson [www.codemagus.com].
   *
   * Copyright (c) 2008 Code Magus Limited. All rights reserved.
   */

  /*
   * $Author: hayward $
   * $Date: 2010/02/01 14:05:39 $
   * $Id: recio.h,v 1.16 2010/02/01 14:05:39 hayward Exp $
   * $Name:  $
   * $Revision: 1.16 $
   * $State: Exp $
   *
   * $Log: recio.h,v $
   * Revision 1.16  2010/02/01 14:05:39  hayward
   * Temporarily backout the changes made in
   * Revision 1.14 to the mode enum to add an
   * unset value. This change has held back any
   * changes to recio since August 2009 as it
   * has not been possible to build at the head
   * on all systems since then (esp MVS). This
   * change does require all platforms to be
   * built so that remoteAM works. This change
   * can be implemented in the next build as
   * all the buils platforms are operational
   * again.
   *
   * Revision 1.15  2010/01/27 20:04:24  hayward
   * Add CVS strings.
   *
   * Revision 1.14  2009/08/05 09:37:52  hayward
   * Modify the recio open modes so that RECIO_MODE_SEQ_INPUT
   * does not have the value of 0. This is a problem when a
```

```
   * field of this type is tested for this value in newly NULL
   * initialised storage. The test should fail. This enum now
   * has the value of 1.
   *
   * Revision 1.13  2009/01/12 16:23:44  hayward
   * Change contact information.
   *
   * Revision 1.12  2008/11/01 12:12:07  hayward
   * Add functionality to allow a Recio caller to supply a
   * routine address to call and a parm to use when an Access Method
   * requires the caller to flush all output.
   *
   * Revision 1.11  2008/03/31 22:20:07  stephen
   * Add support for environment variables in AMD files
   *
   * Revision 1.10  2008/03/28 19:32:27  hayward
   * Fix syntax error from MVS port.
   *
   * Revision 1.9  2008/03/11 17:48:07  stephen
   * Expose max message and key length constants to access method interface
   *
   * Revision 1.8  2008/03/11 17:37:13  stephen
   * Expose constants for max message and key length
   *
   * Revision 1.7  2008/02/14 19:23:20  stephen
   * Add support for AMD supported modes and checks and skipseq mode
   *
   * Revision 1.6  2008/02/08 04:52:03  stephen
   * Fix typo in doc and remove unwanted file
   *
   * Revision 1.5  2008/02/07 13:53:11  stephen
   * Clean header and add documentation
   *
   * Revision 1.4  2008/02/07 09:47:30  stephen
   * Fix problems encountered with testprog.c
   *
   * Revision 1.3  2008/02/05 15:53:22  stephen
   * Add new files to module
   *
   * Revision 1.2  2008/01/28 16:29:18  stephen
   * Additional development
   *
   * Revision 1.1.1.1  2008/01/28 15:02:57  stephen
   * Initail import of sources
   *
   */
static char *cvs_recio_h =
   "$Id: recio.h,v 1.16 2010/02/01 14:05:39 hayward Exp $";

  /*
   * Constants and options:
   */
```

```
  /*
   * Exposed types and structures:
   */

typedef enum
   {
#if 0
   RECIO_MODE_UNSET,      /* the stream mode has not been set */
#endif
   RECIO_MODE_SEQ_INPUT,  /* the stream is opened in sequential input mode */
   RECIO_MODE_SEQ_OUTPUT, /* the stream is opened in sequential output mode */
   RECIO_MODE_DIR_INPUT,  /* the stteam is opened in direct input mode */
   RECIO_MODE_DIR_OUTPUT, /* the stream is opened in direct output mode */
   RECIO_MODE_SKIP_INPUT, /* the stream is opened in skip seq input mode */
   RECIO_MODE_NUM_ENTRIES /* number of defined modes */
   } recio_modes_t;

typedef struct recio_stream recio_stream_t;  /* Record I/O Stream Descriptor */

  /* recio_stream_flush_t is a user supplied function which will allow an access
   * method to request that the caller flush any and all pending data still to
   * be written to recio; in other words to flush any data held in the callers
   * buffers.
   * After a recio_open the address of this routine can be saved in the stream
   * member flush.
   */
typedef void (*recio_stream_flush_t)(recio_stream_t *);

struct recio_stream
   {
   unsigned long flags;

#define RECIO_OPT_HELP    0x80000000  /* invoked assistant for open string */
#define RECIO_OPT_VERBOSE 0x40000000  /* process with maximum message output */

   char *open_string;            /* copy of open paramaeter used in open */
   char *last_error;             /* error message returned by recio_error */
   void *stream_data;            /* recio library private data */
   void *access_data;            /* access method module private data */
   recio_stream_flush_t flush;   /* routine for recio to call for flush */
   void *flushparm;              /* Caller saved parm for flush */
   };

  /*
   * Exported functions:
   */

  /* Function recio_error() returns a message relating to the last error
   * returned by one of the other function of the recio library. If the
   * stream is supplied as NULL, then the last error returned unrelated to
   * a stream is returned (for example, in the situation that an recio_open
```

```
   * failed to create a stream, then the error message is placed in a global
   * structure. Otherwise for stream related errors then error message is
   * taken from the stream structure.
   */

char *recio_error(recio_stream_t *stream);

  /* Function recio_open() opens a file object by processing the open string
   * and the open mode. If the open succeeds then a recio_stream_t structure
   * is returned which is to be used on all other operations. If an error
   * occurs then NULL is returned and recio_error_msg contains a fromatted
   * error detailing the error.
   */

recio_stream_t *recio_open(char *open_string, recio_modes_t mode,
      unsigned long flags);

   /* This function closes the stream indicated by the stream handle supplied
    * and frees all resources associated with the stream object.
    */

int recio_close(recio_stream_t *stream);

  /* The recio_assist() funcion is called to build a valid open_string
   * that can be passed to recio_open(). This is useful if the application
   * is hosted on a machine which allows an operated to be guided to select
   * the correct method and object for the opening of the file.
   */

char *recio_assist(void);

  /* Function recio_read() reads the next record from the input record stream.
   * The stream is expected to be in the correct state for reading.
   * If the read succeeds, then the function returns the length of the record
   * placed in the parameter buf. This length will be less than or equal to the
   * supplied value of the parameter len. If the end of the stream is
   * encountered during processing then the function will return -1 to indicate
   * that there is no more data in the stream and a call to the recio_eof()
   * function will return a value of 1. In the direct mode, this also
   * indicates that the current record pointer does not refer to a valid
   * record (a not found condition) and in this case the recio_notfound()
   * function will return 1. If an error occurs which is determined
   * to be non-recoverable and/or which the application has to deal with,
   * then the function also returns -1.
   */

int recio_read(recio_stream_t *stream, int len, unsigned char *buf);

  /* Function recio_write() writes a record to the output stream.
   * The stream is expected to be in the correct state for writing reading.
   * If the write succeeds, then the function returns the length of the record
   * found in buf. This length of this record is indicated by the len
```

```
   * parameter. If an error occurs during the processing of the record, then
   * -1 is returned by the function. In the direct mode if there is no
   * current key value set or a duplicate record is detected then the
   * function will return 0.
   */

int recio_write(recio_stream_t *stream, int len, unsigned char *buf);

   /* Function recio_point() takes the key or address of the next record
    * when processing the file in random manner and sets that key or
    * address as the next record to read or write. If the operation is
    * successful, then it will return 0. Otherwise -1 is returned and a
    * message is placed in the stream structure and which can be extracted
    * using the recio_error() function.
    */

int recio_point(recio_stream_t *stream, int len, unsigned char *buf);

  /* Function recio_tell() returns the key or address of the last record
   * returned. If the function is successful, then the length of the
   * key or address is returned. Otherwise -1 is returned and a message
   * is placed in the supplied stream structure and which can be extracted
   * using the recio_error() function.
   */

int recio_tell(recio_stream_t *stream, int len, unsigned char *buf);

  /* Function recio_eof() returns 1 if the given stream has been used for
   * input operations and is at the end of the stream. Otherwise the function
   * returns 0. A call to recio_eof should be made to check against the eof
   * condition after a recio_read operation has been attempted. If the read
   * operation fails to return data from the call, it will return -1 to
   * indicate an unusual condition. In this case, for sequential input the
   * eof condition should be used to check whether this is the cause.
   */

int recio_eof(recio_stream_t *stream);

  /* Function recio_key2string() converts a given key into a string
   * format suitable for displaying. The display format of the key
   * should make sense with respect to the underlying object being
   * access through the library. The function returns the length of
   * the key if successful and returns -1 if the function fails and
   * places a message in the the supplied stream structure and which
   * can be extracted by the recio_error() function.
   */

int recio_key2string(recio_stream_t *stream, int len, unsigned char *buf,
      char *key_string);

  /* Function recio_string2key() converts a string format key into a
   * key or record address suitable for procesing by the underyling
```

```
   * access method against the object. If the conversion is succesful
   * then the function returns the length of the key. Otherwise -1
   * is returned and a message is place in the supplied stream structure
   * and which can be extrected by the recio_error() function.
   */

int recio_string2key(recio_stream_t *stream, int len, unsigned char *buf,
      char *key_string);

  /*
   * Globals variables exported by library:
   */

#ifdef RECIO_INCLUDED_FROM_RECIO_C
   #define RECIO_EXTERN /* local */
#else
   #define RECIO_EXTERN extern
#endif


#undef RECIO_INCLUDED_FROM_RECIO_C
#undef RECIO_EXTERN

#endif /* RECIO_H */
```

4.1 The `recio_error` Function

```
char *recio_error(recio_stream_t *stream);
```

Function `recio_error()` returns a message relating to the last error returned by one of the other function of the recio library. If the stream is supplied as NULL, then the last error returned unrelated to a stream is returned (for example, in the situation that an `recio_open` failed to create a stream, then the error message is placed in a global structure. Otherwise for stream related errors then error message is taken from the stream structure.

4.2 The `recio_open` Function

```
recio_stream_t *recio_open(char *open_string, recio_modes_t mode,
         unsigned long flags);
```

Function `recio_open()` opens a file object by processing the open string and the open mode. If the open succeeds then a `recio_stream_t` structure is returned which is to be used on all other operations. If an error occurs then NULL is returned and `recio_error_msg` contains a formatted error detailing the error.

The open specification string passed to the `recio_open` function in the `open_string`

parameter supplies three pieces of information to the `recio` library for the establishment of a record stream:

4.2.1 The Access Method Specification

The access method specification is supplied in the open specification string as the first *identifier* of the string. In the example in Section 4.11 on page 15 the access method specification is the portion of the string

```
sample(/tmp/in,recfm=v,mode=rb)
```

that reads `sample`. This specifies that the access method `sample` will be used to process the underlying object of the stream. The attributes of this access method are specified in the corresponding access method definition file `SAMPLE.amd`.

4.2.2 Object Specification

The next portion of the open specification string determines the object that the underlying access method is to process. In the sample from Section 4.11 on page 15, these are, respectively, the files `/tmp/in` and `/tmp/out`.

The object is specified as being the text from the first open parenthesis `(` following the access method name up to, but excluding the un-nested comma or close parenthesis (in the case that there are no access method options specified). The nesting referred to here is nesting within pairs of open and close brackets ("`[`" and "`]`"). If the object specification starts with and ends with a bracket, these do not form part of the object specification.

4.2.3 Access Method Options

The final part of the open specification string is a list of access method open options. In our example, these are the portions that read `recfm=v,mode=rb` and `recfm=v,mode=rb`.

Options are name-value pairs where the name must conform to an identifier and the value may be a string, integer, identifier such as a file name, or a sequence of characters bracketed using the left and right square bracket characters ("`[`" and "`]`"). The formats of the file names are somewhat restrictive, but an arbitrary value can be accommodated by expressing it as the content of a string. When a value is placed between "`[`" and "`]`" characters, the text within the outermost pair of brackets forms the value, but any bracketing within the outer most brackets must be balanced. A line break may not appear within a value.

The following command using `recio` for input and output processing is an example showing the options enclosed in brackets. This includes an example of where enclosing

the object name in brackets is also useful (the command shown here is split for the convenience of this document):

```
[stephen@blackbox ~]$ cmlcopy -i "remote(
   [mvs([DISP=OLD,DSN=STEPHEN.IMAGE.P00000.D179.T085928,VOL=SER=(T03356),
         UNIT=VSMTAP,LABEL=(1,SL)],using=binary,
         with=[recfm=f,mode=rb,reclen=4096])],
   host=mvs.codemagus.com,user=SECUSER,password=SECPASS,timeout=900)"
   -o "binary(/home/stephen/ADFA2100.image,mode=wb,reclen=4096,recfm=f)"
```

4.2.4 Processing Modes

The `mode` parameter may have one of the values defined in the `enum recio_modes_t`. These values and what they mean are:

- `RECIO_MODE_SEQ_INPUT` The stream is to be opened in sequential input mode. In this mode the stream is positioned at the start of the stream causing the first read operation to return the first record, and the second read operation to retrieve the second record, etc. (if they exist).
- `RECIO_MODE_SEQ_OUTPUT` The stream is to be opened in sequential output mode. In this mode the stream is emptied of all records and positioned at the start of the stream. The first write operation inserts the first record of the stream, the second write operation inserts the second record, etc.
- `RECIO_MODE_DIR_INPUT` The stream is to be opened in direct access input mode. In this mode of operation a point operation preceding a read operation will determine the record to be returned on the point operation (if the point operation succeeded).
- `RECIO_MODE_DIR_OUTPUT` The stream is to be opened in direct access output mode. The position of the next record to be written is determined by a point operation which provides a key for the position of the record.
- `RECIO_MODE_SKIP_OUTPUT` The stream is to be opened in skip sequential processing input mode. The stream is opened in the sequential mode, but the sequence of records returned can be modified using the point operation.

4.2.5 Processing Flags

The `flags` parameter the bit-wise or of the individual flag values:

- `RECIO_OPT_HELP`

  The assistant is to be invoked in order to help with the derivation of a value open string specification.

- RECIO_OPT_VERBOSE

  Processing within the library is as verbose as possible. This is useful for debugging or looking for syntax errors in access method definition files. This option is also in effect if the environment variable RECIO_OPT_VERBOSE is set to the value "1".

4.3 The recio_close Function

```
int recio_close(recio_stream_t *stream);
```

This function closes the stream indicated by the stream handle supplied and frees all resources associated with the stream object.

4.4 The recio_read Function

```
int recio_read(recio_stream_t *stream, int len, unsigned char *buf);
```

Function recio_read() reads the next record from the input record stream. The stream is expected to be in the correct state for reading. If the read succeeds, then the function returns the length of the record placed in the parameter buf. This length will be less than or equal to the supplied value of the parameter len. If the end of the stream is encountered during processing then the function will return −1 to indicate that there is no more data in the stream and a call to the recio_eof() function will return a value of 1. In the direct mode, this also indicates that the current record pointer does not refer to a valid record (a not found condition) and in this case the recio_notfound() function will return 1. If an error occurs which is determined to be non-recoverable and/or which the application has to deal with, then the function also returns −1.

4.5 The recio_write Function

```
int recio_write(recio_stream_t *stream, int len, unsigned char *buf);
```

Function recio_write() writes a record to the output stream. The stream is expected to be in the correct state for writing reading. If the write succeeds, then the function returns the length of the record found in buf. This length of this record is indicated by the len parameter. If an error occurs during the processing of the record, then −1 is returned by the function. In the direct mode if there is no current key value set or a duplicate record is detected then the function will return 0.

4.6 The `recio_point` Function

```
int recio_point(recio_stream_t *stream, int len, unsigned char *buf);
```

Function `recio_point()` takes the key or address of the next record when processing the file in random manner and sets that key or address as the next record to read or write. If the operation is successful, then it will return `0`. Otherwise `-1` is returned and a message is placed in the stream structure and which can be extracted using the `recio_error()` function.

4.7 The `recio_tell` Function

```
int recio_tell(recio_stream_t *stream, int len, unsigned char *buf);
```

Function `recio_tell()` returns the key or address of the last record returned. If the function is successful, then the length of the key or address is returned. Otherwise `-1` is returned and a message is placed in the supplied stream structure and which can be extracted using the `recio_error()` function.

4.8 The `recio_eof` Function

```
int recio_eof(recio_stream_t *stream);
```

Function `recio_eof()` returns 1 if the given stream has been used for input operations and is at the end of the stream. Otherwise the function returns `0`. A call to `recio_eof` should be made to check against the eof condition after a `recio_read` operation has been attempted. If the read operation fails to return data from the call, it will return `-1` to indicate an unusual condition. In this case, for sequential input the eof condition should be used to check whether this is the cause.

4.9 The `recio_key2string` Function

```
int recio_key2string(recio_stream_t *stream, int len, unsigned char *buf,
            char *key_string);
```

Function `recio_key2string()` converts a given key into a string format suitable for displaying. The display format of the key should make sense with respect to the underlying object being access through the library. The function returns the length of the key if successful and returns `-1` if the function fails and places a message in the the supplied stream structure and which can be extracted by the `recio_error()` function.

4.10 The `recio_string2key` Function

```
int recio_string2key(recio_stream_t *stream, int len, unsigned char *buf,
            char *key_string);
```

Function `recio_string2key()` converts a string format key into a key or record address suitable for processing by the underlying access method against the object. If the conversion is successful then the function returns the length of the key. Otherwise `-1` is returned and a message is place in the supplied stream structure and which can be extracted by the `recio_error()` function.

4.11 Sample Program Using the `recio` Record Stream I/O Library

The following worked example illustrates the usage of the library by a program. In this example, assuming that the program was compiled using the command:

```
cc -g -I. -I../include -L. -L../lib ${COPT} -o testprog testprog.c \
      -lpopt recio.o -ldl -lrecio
```

Then, entering the following from the command line, executes the program:

```
./testprog -i "sample(/tmp/in,recfm=v,mode=rb)" \
           -o "sample(/tmp/out,recfm=v,mode=wb)"
```

```
  /* File: testprog.c
   *
   * This program is the test program for the recio library. This libray
   * performs I/O using record streaming. The access method used in the
   * record streaming is determined by an access method named corresponding
   * to an access method definition file. It is in this file that the access
   * method module is named and the parameters or option requirements are
   * stipulated.
   *
   * Author: Stephen R. Donaldson [www.codemagus.com]
   *
   * Copyright (c) 2008 Code Magus Limited. All rights reserved.
   *
   */

  /*
   * $Author: hayward $
   * $Date: 2009/10/28 13:11:29 $
   * $Id: testprog.c,v 1.23 2009/10/28 13:11:29 hayward Exp $
   * $Name:  $
   * $Revision: 1.23 $
   * $State: Exp $
   *
   * $Log: testprog.c,v $
   * Revision 1.23  2009/10/28 13:11:29  hayward
```

```
* Add sequential only flag to testprog.
* Improve diagnostic output when reread record does
* not match by dumping the records with dumpbuff.
*
* Revision 1.22  2009/09/17 17:29:21  hayward
* Add the ability to continue reading records once
* a read error occurs. This enables a tester to test
* tell and direct read even after a previous read error.
*
* Revision 1.21  2009/03/24 08:46:54  hayward
* Update comments.
*
* Revision 1.20  2009/03/24 08:16:58  hayward
* Close the files after the sequential read process. Then if a re-read of
* a specific key is requested, re-open the file, call point, read and close.
* This mimics the behaviour of the FIP and tests the bug in remoteam that
* did an invalid check on the length of the key if a point was done before
* a tell on an open stream.
*
* Revision 1.19  2009/02/10 10:57:52  hayward
* Check that recio returns correct value for key2string().
*
* Revision 1.18  2009/02/06 15:22:29  hayward
* Chnage testprog - so that the only library it depends on is
* recio (and any it depnds on - so far only hashtab). This makes
* it available to distribute as a sample program with the package.
*
* Revision 1.17  2008/10/01 09:32:59  hayward
* Add more output to error messages. This helps if debugging
* an AM that does not give a message - this will now be very
* noticeable.
*
* Revision 1.16  2008/09/29 13:08:59  hayward
* Allow for multiple iterations of open-read/write-close.
*
* Revision 1.15  2008/07/15 15:49:04  hayward
* Add max records to read and clean up saved key processing.
*
* Revision 1.14  2008/05/14 16:10:22  hayward
* Timestamp function moved to osmods.
*
* Revision 1.13  2008/05/06 16:41:49  stephen
* Add access method utility function get_opt_string
*
* Revision 1.12  2008/04/14 09:57:10  stephen
* Allow testprog compile on Windows
*
* Revision 1.11  2008/04/09 16:21:52  hayward
* Added code to test only sequential access methods
*
* Revision 1.10  2008/04/08 09:49:04  hayward
* Add ability to vary number of records read before saving key.
```

```
   * If verbose is on also print timestamps - so that progress
   * can be mapped based on each time a key is saved.
   *
   * Revision 1.9  2008/03/28 18:38:02  hayward
   * If there are less than 1000 records then save the key of the
   * 3rd record rather than the 1st as this may be a better test.
   *
   * Revision 1.8  2008/03/21 23:05:32  hayward
   * Improve error checking.
   *
   * Revision 1.7  2008/03/11 15:51:04  hayward
   * Save key of first record - allows any number of
   * records to be tested.
   * Guard key processing against reading no records.
   * Output number of records read.
   *
   * Revision 1.6  2008/03/11 13:03:53  hayward
   * Change default name of yy_ functions.
   *
   * Revision 1.5  2008/02/14 19:23:20  stephen
   * Add support for AMD supported modes and checks and skipseq mode
   *
   * Revision 1.4  2008/02/14 17:44:44  stephen
   * Changes and checks for skip sequential processing
   *
   * Revision 1.3  2008/02/08 10:22:46  stephen
   * Changes to support point and tell in standard access method
   *
   * Revision 1.2  2008/02/07 09:47:30  stephen
   * Fix problems encountered with testprog.c
   *
   * Revision 1.1  2008/02/07 06:12:35  stephen
   * Add all new files to module
   *
   */

static char *cvs_testprog_c =
   "$Id: testprog.c,v 1.23 2009/10/28 13:11:29 hayward Exp $";

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <popt.h>

#include <osmods.h>
#include <recio.h>
#include <dumpbuff.h>

  /*
   * Local prototypes.
```

```
   */

  /*
   * Command line parsing popt data structure and option variables:
   */

char *input_spec = NULL;        /* input file recio specfication */
char *output_spec = NULL;       /* output file recio specification */
int records_per_key = 10000;    /* Save and report key of every nth record */
int max_records = 0;            /* Maximum records to read; 0=all */
int max_iterations = 1;         /* Maximum number of times to open files, read
                                   and write records and close files. */
int max_read_errors = 0;        /* Maximum number of read errors to tolerate in
                                   reading sequentially. */
int sequential_only = 0;        /* Process input in sequential mode only - do not
                                   attmept to use skip sequential */
int verbose = 0;                /* flag for switching on verbose processing */

struct poptOption optionsTable[] =
   {
      {  "input", 'i', POPT_ARG_STRING, &input_spec, 0,
         "Input File Specification in record stream format",
         "<access-method>(<object-name>[,<options>])"},
      {  "output", 'o', POPT_ARG_STRING, &output_spec, 0,
         "Output File Specification in record stream format",
         "<access-method>(<object-name>[,<options>])"},
      {  "records-per-key", 'r', POPT_ARG_INT, &records_per_key, 0,
         "Report and save key (for reread) of every nth record",
         "{10000|<number>}"},
      {  "Iteration loop amount", 'l', POPT_ARG_INT, &max_iterations, 0,
         "Perform \"open - read - write - close\" this many times",
         "{1|<number>}"},
      {  "max-records", 'm', POPT_ARG_INT, &max_records, 0,
         "Maximum records to read (0 = all)", "{0|<number>}"},
      {  "max-read-errors", 'e', POPT_ARG_INT, &max_read_errors, 0,
         "Maximum read errors that can be tolerated", "{0|<number>}"},
      {  "sequential-only", 's', POPT_ARG_NONE, &sequential_only, 0,
         "Do not attempt to open input in skip sequential mode", NULL},
      {  "verbose", 'v', POPT_ARG_NONE, &verbose, 0,
         "Verbose processing mode", NULL},
         POPT_AUTOHELP
      { NULL, 0, 0, NULL, 0 }
   };

int main(int argc, char *argv[])
   {
   poptContext optCon;           /* for processing command lince arguments */
   recio_stream_t *input = NULL; /* record input stream */
   recio_stream_t *output;       /* record output stream */
   unsigned long flags;          /* record stream I/O library flags */
   int len;                      /* record length */
   unsigned char buf[65536];     /* data record */
```

```
 unsigned char key[200];       /* record key */
 char key_string[200];         /* record key as a string */
 char save_key[200];           /* key of last record of tell */
 char save_buf[65536];         /* buf of last record of tell */
 int save_len = 0;             /* len of last record of tell, 0 if none */
 int recno;                    /* record counter */
 int rc;
 int input_skip = 0;           /* Does input support skip read */
 long long bytes_read = 0;     /* data bytes read */
 long long bytes_written = 0;  /* data bytes written */
 int iteration;                /* open-read/write-close iteration number */
 int keylen;                   /* Returned keylength from recio. */
 int read_errors = 0;          /* number of read errors encountered */

/* Process the command line options and populate the command line option
 * variables.
 */
 optCon = poptGetContext(argv[0],argc,(const char **)argv,optionsTable,0);
 rc = poptGetNextOpt(optCon);
 if (rc < -1)
    {
    fprintf(stderr,"%s\n",poptStrerror(rc));
    poptPrintUsage(optCon, stderr, 0);
    exit(16);
    }

/* Make sure that the mandatory command line arguments have been provided.
 */
 if (!input_spec || !output_spec)
    {
    fprintf(stderr,
      "--input-file-spec and --output-file-spec arguments are mandatory.\n");
    poptPrintUsage(optCon, stderr, 0);
    exit(16);
    }

/* Print start message:
 */

/* Setup the flags for opening the streams:
 */
 flags = 0;
 if (verbose) flags |= RECIO_OPT_VERBOSE;

 for (iteration=1; iteration<=max_iterations; iteration++)
    {
   /* Attempt to open the input and output record streams.
    */
    if (!sequential_only)
       input = recio_open(input_spec,RECIO_MODE_SKIP_INPUT,flags);
    if (!input)
       {
```

```
   if (!sequential_only)
      {
      fprintf(stderr,"Attempt to open input in SKIP mode failed:\n\n");
      fprintf(stderr,"%s\n",recio_error(input));
      fprintf(stderr,"\nWill now attempt SEQential mode.\n");
      }

  /* Attempt to open it in normal mode.
   */
   input = recio_open(input_spec,RECIO_MODE_SEQ_INPUT,flags);
   if (!input)
      {
      fprintf(stderr,"Attempt to open input in SEQ mode failed:\n");
      fprintf(stderr,"%s\n",recio_error(input));
      poptPrintUsage(optCon, stderr, 0);
      exit(16);
      }
   }
else
   input_skip = 1;

output = recio_open(output_spec,RECIO_MODE_SEQ_OUTPUT,flags);
if (!output)
   {
   fprintf(stderr,"Open output error: %s\n",recio_error(output));
   poptPrintUsage(optCon, stderr, 0);
   exit(16);
   }

/* Read the records from the input stream, writing then out to the output
 * stream.
 */
recno = 0;
while (recno < max_records || max_records == 0)
   {
  /* Attempt to read the next input record. If the stream is at the end
   * then break out of the loop.
   */
   rc = recio_read(input,sizeof(buf),buf);
   if (rc && recio_eof(input)) break;

  /* Check for any operation errors on the stream: If there is an error
   * and it is allowed then continue but do not write the record.
   * If the max errors allowed has been surpassed then just break out of
   * the loop to attempt the reread once all the sequential reads are
   * complete.
   */
   recno++;
   if (rc < 0)
      {
      fprintf(stderr,"read error: %s\n",recio_error(input));
      read_errors++;
```

```
         if (read_errors > max_read_errors) break;
         }
      else
         {
        /* Write the record to the output stream.
         */
         bytes_read += rc;
         len = rc;
         rc = recio_write(output,len,buf);

        /* Check for any operation errors on the stream:
         */
         if (rc < 0)
            {
            fprintf(stderr,"write error: %s\n",recio_error(output));
            poptPrintUsage(optCon, stderr, 0);
            exit(16);
            }
         bytes_written += rc;
         }

      /* Display the position of every record'th record:
       */
       if (input_skip && recno % records_per_key == 0)
         {
         rc = recio_tell(input,sizeof(key),key);
         if (rc > 0)
            {
            keylen = recio_key2string(input,rc,key,key_string);
            fprintf(stderr,"Key of record %d = %s. ", recno,key_string);
            fprintf(stderr,"Returned/Calculated length %d/%d.\n",keylen,
                  strlen(key_string));

           /* Save the key and the record:
            */
            strcpy(save_key,key_string);
            save_len = len;
            memcpy(save_buf,buf,len);
            }
         else
            fprintf(stderr,"tell error: %s\n",recio_error(input));
            }
      } /* while */

   /* Now close the files.
    */
    recio_close(input);
    recio_close(output);

   /* Go back to the last saved record and attempt to re-read that record. The
    * check is to compare the original record to the re-read record.
    * This behaviour mimics the FIP in that it reads n records sequentially
```

```
* and at the same time saves the record keys. It then closes the file and
* as the user requests details of records it re-opens the file, does a
* point for the record and reads the data.
*/
if (input_skip && save_len)
   {
   fprintf(stderr,
         "Re-opening input to re-read record %s for comparison\n",
         save_key);
   input = recio_open(input_spec,RECIO_MODE_SKIP_INPUT,flags);
   if (!input)
      {
      fprintf(stderr,"Attempt to re-open input in SKIP mode failed:\n\n");
      fprintf(stderr,"%s\n",recio_error(input));
      fprintf(stderr,"%s\n","No checking of direct read can be done");
      }
   else
      {
      memset(key,0,sizeof(key));
      if (verbose)
         fprintf(stderr,"Re-reading record with key = %s.\n", save_key);

      rc = recio_string2key(input,sizeof(key),key,save_key);
      if (rc < 0)
         {
         fprintf(stderr,"string2key error: %s\n",recio_error(input));
         exit(16);
         }

      rc = recio_point(input,rc,key);
      if (rc < 0)
         {
         fprintf(stderr,"point error: %s\n",recio_error(input));
         exit(16);
         }
      fprintf(stderr,"point using saved key=%s, RC=%d\n",save_key,rc);

      rc = recio_read(input,sizeof(buf),buf);
      if (rc < 0)
         {
         fprintf(stderr,"read(2) error: %s\n",recio_error(input));
         exit(16);
         }

      if (rc != save_len)
         {
         fprintf(stderr,
               "Original and re-read lengths mismatch: Was %d, now %d.\n",
               save_len,rc);
         if (verbose)
            {
            fprintf(stderr,"Original record dump.\n");
```

```
                fdumpbuffer(stderr,save_len,save_buf);
                fprintf(stderr,"Re-read record dump.\n");
                fdumpbuffer(stderr,rc,buf);
                }
             }
          else
             if (memcmp(save_buf,buf,save_len))
                {
                fprintf(stderr,"Original and re-read records mismatch.\n");
                if (verbose)
                   {
                   fprintf(stderr,"Original record dump.\n");
                   fdumpbuffer(stderr,save_len,save_buf);
                   fprintf(stderr,"Re-read record dump.\n");
                   fdumpbuffer(stderr,rc,buf);
                   }
                }
             else
                fprintf(stderr,"Original and re-read records match OK.\n");

          recio_close(input);
          }
       }

    /* Print ending messages:
     */

     if (verbose) fprintf(stderr, "Bytes read/written %llu, %llu\n",
           bytes_read, bytes_written);
     fprintf(stderr, "%d records read\n", recno);

     } /* iteration<=max_iterations */

exit(0);
} /* main */
```

5 `recio` Access Method Call Reference

The `recio` library supplies an API for the implementation of access methods. An access method is responsible for the underlying record stream I/O required by the `recio` library. The open specification string passed to the `recio_open` specifies access method definition that is to be used to process the underlying stream. The access method definition in turn specifies the module that implements the access method. The interface described here is the provider interface that the access method needs to support in order to implement an access method to the `recio` library.

An access method module makes the interface structures and prototypes available by

including the header file `reciol.h`. This header in turn includes the header `recio.h` making the stream structures available to the access method library.

```
#ifndef RECIOL_H
#define RECIOL_H
  /* File: reciol.h
   *
   * This header describes the internal data structures used to support the
   * Code Magus Limited Record Stream I/O Module. These structures are passed
   * between the recio library and a loaded supporting module. Any service
   * functions which the recio library in turn makes available to the
   * loaded access module are made available through these structures.
   *
   * This header is not for distribution.
   *
   * Copyright (c) 2008 Code Magus Limited. All rights reserved.
   *
   * Author: Stephen R. Donaldson [www.codemagus.com].
   */


  /*
   * $Author: hayward $
   * $Date: 2010/02/01 14:12:42 $
   * $Id: reciol.h,v 1.14 2010/02/01 14:12:42 hayward Exp $
   * $Name:  $
   * $Revision: 1.14 $
   * $State: Exp $
   *
   * $Log: reciol.h,v $
   * Revision 1.14  2010/02/01 14:12:42  hayward
   * Temporarily backout the changes made in
   * Revision 1.14 to the mode enum to add an
   * unset value. This change has held back any
   * changes to recio since August 2009 as it
   * has not been possible to build at the head
   * on all systems since then (esp MVS). This
   * change does require all platforms to be
   * built so that remoteAM works. This change
   * can be implemented in the next build as
   * all the buils platforms are operational
   * again.
   *
   * Revision 1.13  2010/01/27 20:04:24  hayward
   * Add CVS strings.
   *
   * Revision 1.12  2009/08/05 09:37:52  hayward
   * Modify the recio open modes so that RECIO_MODE_SEQ_INPUT
   * does not have the value of 0. This is a problem when a
   * field of this type is tested for this value in newly NULL
   * initialised storage. The test should fail. This enum now
   * has the value of 1.
   *
```

```
   * Revision 1.11  2009/01/12 16:23:44  hayward
   * Change contact information.
   *
   * Revision 1.10  2008/07/28 15:43:00  hayward
   * Add access method utility function get_access_name
   * Fix bug in error message showing available modes.
   *
   * Revision 1.9  2008/05/06 16:41:49  stephen
   * Add access method utility function get_opt_string
   *
   * Revision 1.8  2008/03/31 22:20:07  stephen
   * Add support for environment variables in AMD files
   *
   * Revision 1.7  2008/03/11 17:48:07  stephen
   * Expose max message and key length constants to access method interface
   *
   * Revision 1.6  2008/02/14 19:23:20  stephen
   * Add support for AMD supported modes and checks and skipseq mode
   *
   * Revision 1.5  2008/02/07 09:47:30  stephen
   * Fix problems encountered with testprog.c
   *
   * Revision 1.4  2008/02/07 05:40:19  stephen
   * Dev updates and add lex and yacc files
   *
   * Revision 1.3  2008/02/05 15:53:22  stephen
   * Add new files to module
   *
   * Revision 1.2  2008/01/28 16:29:18  stephen
   * Additional development
   *
   * Revision 1.1  2008/01/28 15:05:39  stephen
   * Rename of reciop.h to reciol.h
   *
   * Revision 1.1.1.1  2008/01/28 15:03:02  stephen
   * Initail import of sources
   *
   */
static char *cvs_reciol_h =
   "$Id: reciol.h,v 1.14 2010/02/01 14:12:42 hayward Exp $";

#include <hashtab.h>

#include "recio.h"

  /*
   * Options and constants:
   */

#define RECIO_MAX_MODES    20      /* size of access method modes array */
#define RECIO_MSG_MAX 4000         /* max possible message length */
#define RECIO_KEY_MAX 32760        /* max possible key length */
```

```
static char *recio_mode_names[RECIO_MODE_NUM_ENTRIES] =
   {
#if 0
   "RECIO_MODE_UNSET",      /* the stream mode has not been set. */
#endif
   "RECIO_MODE_SEQ_INPUT",  /* the stream is opened in sequential input mode */
   "RECIO_MODE_SEQ_OUTPUT", /* the stream is opened in sequential output mode */
   "RECIO_MODE_DIR_INPUT",  /* the stteam is opened in direct input mode */
   "RECIO_MODE_DIR_OUTPUT", /* the stream is opened in direct output mode */
   "RECIO_MODE_SKIP_INPUT", /* the stream is opened in skip seq input mode */
   };

  /*
   * Types and structures:
   */

  /* Functions expected to be implemented by the access method loaded
   * by the recio library are expected to be populated into the stream
   * structure for routing calls back to the library.
   */

typedef struct recio_access recio_access_t;  /* Access method structure */

  /* The access method is expected to support an initialisation function in
   * which the access method functions supported are populated and to which
   * flags are passed indicating which methods are expected to be implemented
   * by the loaded access method. If the required methods cannot be implemented
   * then the function must fail and provide a method in the last_error
   * field of the access structure. A failed initialisation is indicated by
   * returning a -1 from the call. Otherwise zero is returned to indicate
   * success.
   */

typedef int (*recio_access_init_t)(recio_access_t *access, unsigned long flags);

  /* Interface utility functions are those functions provided by the
   * recio library for the usage by the access method modules. These
   * are utility functions which can be expanded on as required.
   */

  /* The recio_option_lookup_t is the typedef of the access utililty
   * function supplied by the recio library and allows the access method
   * modules to lookup values supplied in the name-value pairs in the
   * open string and/or defaulted by the access method definition. If the
   * name is not valid then a NULL is returned an error is placed in the
   * last_error member of the stream structure. A variable may not have a
   * value, and in this case an emty string is returned (effectively an
   * option always has a default value). Otherwise the open or default
   * value is returned.
   */
```

```
typedef char *(*recio_option_lookup_t)(recio_stream_t *stream, char *name);

  /* String extracting functions where it it is clear from the the actual
   * utility what the string attribute required is are described using the
   * recio_get_string_t type.
   */

typedef char *(*recio_get_string_t)(recio_stream_t *stream);

  /* Status chaging functions have a type of recio_set_status_t. These
   * functions set the indicated condition in the stream. These utility
   * functions are used by access methods to communicate the processing
   * status back to the caller through the recio library.
   */

typedef int (*recio_set_status_t)(recio_stream_t *stream);

  /* The access method is expected to support an open function which
   * initialises the structure to and supplies the object name being
   * opened to the library. A sucess object open returns a zero from
   * the function and the access should be in a state ready for the
   * first operation on the stream. If the operation fails, then it
   * is expected to return -1 to the caller and to place a meaningful
   * message in the last_error field of the stream structure.
   */

typedef int (*recio_access_open_t)(recio_access_t *access,
      recio_stream_t *stream, char *object, recio_modes_t mode,
      unsigned long flags);

  /* The access method is expected to support an close function which
   * frees all resource obtained by the access method and relinquishes
   * any locks or control on the underlying object. If the operation
   * fails then the function should return -1 and place a meaningful
   * message in the last_error field of the stream structure.
   */

typedef int (*recio_access_close_t)(recio_access_t *access,
      recio_stream_t *stream);

  /* The access method is expected to support a read function which moves
   * the next record (sequential read) or the last record pointed to
   * (after a point operation) into the supplied buffer. On a successful read
   * the function returns the number of bytes actually read and on failure
   * the function should return -1 and place a meaningful message in the
   * last_error field of the stream structure.
   */

typedef int (*recio_access_read_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);

  /* The access method is expected to support a write function which moves
```

```
   * the next record to the underlying object. On successful write the function
   * should return the number of bytes logically written (that is the number
   * of bytes should be the same as the length of the supplied buffer)
   * and on a failed write the function should return -1 and place a message
   * in the last_error field of the stream structure.
   */

typedef int (*recio_access_write_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);

  /* The access method may supply methods to facilitate random access to the
   * underlying object. If this is the case and the access mode is random then
   * a read operation may be preceded by a point operation in which the key
   * of the record, address of the record or offset of the record may be given.
   * If the function should fail then -1 must be returned and a message placed
   * in the last_error field of the stream structure.
   */

typedef int (*recio_access_point_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);

  /* The access method may supply methods to facilitate random access to the
   * underlying object or to provide an identity of the records read.
   * If this is the case and the access mode is random then an read operation
   * may be followed by a tell operation which returns the key of the record,
   * address of the record, or offset of the record just read.  If the
   * function should fail then -1 should be returned and a meaningful message
   * placed in the last_error field of the stream structure. In the absence
   * of such a method then the records are simply assigned sequnce numbers
   * and referred to as sequence numbers.
   */

typedef int (*recio_access_tell_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);

  /* For random process and/or for record identification the supplied keys
   * are expected to be in a non-display internal form. For the purposes
   * of reporting and showing record addresses and keys in a display form,
   * the access may provide a set of transfer functions which convert a
   * supplied key/address of a record into a display form and back into the
   * internal form. In the absence of such a function, the recio library
   * will supply a set of methods which simply convert the string into the
   * character hexadecimal representation of the key or record address.
   * If the conversion is successful, then the length of the key is returned,
   * otherwise -1 is returned and a message is placed in the last_error
   * structure of the supplied stream.
   */

typedef int (*recio_key2string_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf, char *key_string);
typedef int (*recio_string2key_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf, char *key_string);
```

```
struct recio_access
   {
   char *name;                    /* name of the access method in def   */

  /* Interface utlity functions. These are functions that the access method
   * can call in order to get services from the recio library.
   */
   recio_option_lookup_t option_lookup;    /* return item value */
   recio_get_string_t get_opt_string;      /* return full options string */
   recio_get_string_t get_access_name;     /* return access name */
   recio_set_status_t set_eof;             /* mark stream as end of file */
   recio_set_status_t set_notfound;        /* indicate record not found */

  /* Address of the access functions. These addresses are supplied by
   * access methods initialisation function. This function is called
   * only on the first reference the access method. All other references
   * to the access method through the recio stream will use the same
   * structure.
   */
   recio_access_open_t open;       /* open an object for record I/O      */
   recio_access_close_t close;     /* release object and resources       */
   recio_access_read_t read;       /* read next record from the object   */
   recio_access_write_t write;     /* write next record to the object    */
   recio_access_point_t point;     /* supply next record key/address     */
   recio_access_tell_t tell;       /* extract last record key or address */
   recio_key2string_t key2string; /* convert a key/address to a string  */
   recio_string2key_t string2key; /* convert a string to a key/address  */

  /* An access method can have private data. The private data is only
   * referred to by the access method. Notice that there is no opportunity
   * to release any access method owned private data as the access method
   * code is never released.
   */
   void *private_data;             /* access method private data         */

  /* Any error processing at an access method level needs to place an error
   * message in the last_error field of the access structure.
   */
   char *last_error;               /* for reporting errors access errors */

  /* The attributes loaded from the access method definition are inserted
   * into the access method structure during the parsing of the access
   * method definition.
   */
   int num_modes;                 /* Number of modes supported */
   recio_modes_t modes[RECIO_MAX_MODES];  /*  supported modes */
   hashtab_t *options;            /* options-name/value pairs from access def */
   unsigned long methods;         /* flag for implemented methods */

#define RECIO_METHOD_OPEN     0x80000000   /* implements open */
```

```
#define RECIO_METHOD_CLOSE    0x40000000   /* implements close */
#define RECIO_METHOD_READ     0x20000000   /* implements read */
#define RECIO_METHOD_WRITE    0x10000000   /* implements write */
#define RECIO_METHOD_POINT    0x08000000   /* implements point */
#define RECIO_METHOD_TELL     0x04000000   /* implements tell */

   hashtab_t *descriptions;     /* descriptions of options */
   hashtab_t *constraints;      /* constraints on option values */
   hashtab_t *envvars;          /* envars to be set on open */
   hashtab_t *regexs;           /* compiled regexs of the constraints */
   char *path;                  /* path statement for access method module */
   char *module;                /* name of access method module */
   char *entry;                 /* entry point of initialisation function */
   };


  /*
   * Exported Functions:
   */


#endif /* RECIOL_H */
```

5.1 Stream Structure

The stream structure (`recio_stream_t`) describes a stream instance. The handle of the stream as far as the application program is concerned is the address of a stream structure.

The stream structure provides some information which was made available at the time of the `recio_open` call. These include the open specification string (`open_string`) and the open flags (`flags`).

The structure also includes and string pointer field into which the access method is expected to format details all errors to be communicated back to the user of the `recio` library (`last_error`).

The stream structure also contains an access method usable `stream_data` field for the use by the access method. This is for the access method to store any stream specific private data pertaining to the state of the underlying object being processed.

5.2 Access Structure

All calls to the access method from the `recio` library include the address of the access structure (`recio_access_t`). The access structure is describes the access method and there is one access structure shared amongst all stream instances using the same access method.

The access structure includes the addresses of the functions supported by the access method and are populated by the access method when the access method is initialised (this is done the first time the access method is referred to in an open specification string).

```
/* Address of the access functions. These addresses are supplied by
 * access methods initialisation function. This function is called
 * only on the first reference the access method. All other references
 * to the access method through the recio stream will use the same
 * structure.
 */
 recio_access_open_t open;      /* open an object for record I/O      */
 recio_access_close_t close;    /* release object and resources       */
 recio_access_read_t read;      /* read next record from the object   */
 recio_access_write_t write;    /* write next record to the object    */
 recio_access_point_t point;    /* supply next record key/address     */
 recio_access_tell_t tell;      /* extract last record key or address */
 recio_key2string_t key2string; /* convert a key/address to a string  */
 recio_string2key_t string2key; /* convert a string to a key/address  */
```

The access structure also includes function addresses for functions that are provided by the access method and are utility and callback functions used by the access method during the processing of the access method functions.

```
/* Interface utility functions. These are functions that the access method
 * can call in order to get services from the recio library.
 */
 recio_option_lookup_t option_lookup;    /* return item value */
 recio_set_status_t set_eof;             /* mark stream as end of file */
 recio_set_status_t set_notfound;        /* indicate record not found */
```

5.3 `recio` Library Utility Functions for Access Methods

The above function addresses supply the entry points of functions supplied by the `recio` library for use by access methods for extracting and updating stream state.

5.3.1 The `option_lookup` Function

```
typedef char *(*recio_option_lookup_t)(recio_stream_t *stream, char *name);
```

The `recio_option_lookup_t` is the typedef of the access utility function supplied by the recio library and allows the access method modules to lookup values supplied in the name-value pairs in the open string and/or defaulted by the access method definition. If the name is not valid then a NULL is returned an error is placed in the `last_error` member of the stream structure. A variable may not have a value, and in this case an empty string is returned (effectively an option always has a default value). Otherwise the open or default value is returned.

5.3.2 The `set_eof` Function

```
typedef int (*recio_set_status_t)(recio_stream_t *stream);
```

Status changing functions have a type of `recio_set_status_t`. These functions set the indicated condition in the stream. These utility functions are used by access methods to communicate the processing status back to the caller through the recio library.

5.3.3 The `set_notfound` Function

```
typedef int (*recio_set_status_t)(recio_stream_t *stream);
```

Status charging functions have a type of `recio_set_status_t`. These functions set the indicated condition in the stream. These utility functions are used by access methods to communicate the processing status back to the caller through the recio library.

5.4 Access Method Functions of the Access Method API

The access method functions of the access method API are required to be implemented by the access method module according to the access method definition. If the access method definition does not state that the access method implements the corresponding method the the access method module need not implement the corresponding function.

5.4.1 Access Method Initialisation Function

The access method initialisation function whose name is supplied in the access method definition is the only global entry point that the access method module is required to implement. When the access method is loaded, this entry point is called and the access method is expected to populate the access method function pointers that the access method definition requires to be supported.

5.4.2 Access Method `open` Function

```
typedef int (*recio_access_open_t)(recio_access_t *access,
      recio_stream_t *stream, char *object, recio_modes_t mode,
      unsigned long flags);
```

The access method is expected to support an open function which initialises the structure to and supplies the object name being opened to the library. A success object open

returns a zero from the function and the access should be in a state ready for the first operation on the stream. If the operation fails, then it is expected to return `-1` to the caller and to place a meaningful message in the `last_error` field of the stream structure.

5.4.3 Access Method `close` Function

```
typedef int (*recio_access_close_t)(recio_access_t *access,
      recio_stream_t *stream);
```

The access method is expected to support an close function which frees all resource obtained by the access method and relinquishes any locks or control on the underlying object. If the operation fails then the function should return `-1` and place a meaningful message in the `last_error` field of the stream structure.

5.4.4 Access Method `read` Function

```
typedef int (*recio_access_read_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);
```

The access method is expected to support a read function which moves the next record (sequential read) or the last record pointed to (after a point operation) into the supplied buffer. On a successful read the function returns the number of bytes actually read and on failure the function should return `-1` and place a meaningful message in the `last_error` field of the stream structure.

5.4.5 Access Method `write` Function

```
typedef int (*recio_access_write_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);
```

The access method is expected to support a write function which moves the next record to the underlying object. On successful write the function should return the number of bytes logically written (that is the number of bytes should be the same as the length of the supplied buffer) and on a failed write the function should return `-1` and place a message in the `last_error` field of the stream structure.

5.4.6 Access Method `point` Function

```
typedef int (*recio_access_point_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);
```

The access method may supply methods to facilitate random access to the underlying object. If this is the case and the access mode is random then a read operation may be preceded by a point operation in which the key of the record, address of the record or

offset of the record may be given. If the function should fail then `-1` must be returned and a message placed in the `last_error` field of the stream structure.

5.4.7 Access Method `tell` Function

```
typedef int (*recio_access_tell_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf);
```

The access method may supply methods to facilitate random access to the underlying object or to provide an identity of the records read. If this is the case and the access mode is random then an read operation may be followed by a tell operation which returns the key of the record, address of the record, or offset of the record just read. If the function should fail then `-1` should be returned and a meaningful message placed in the `last_error` field of the stream structure. In the absence of such a method then the records are simply assigned sequence numbers and referred to as sequence numbers.

5.4.8 Access Method `key2string` and `string2key` Function

```
typedef int (*recio_key2string_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf, char *key_string);
typedef int (*recio_string2key_t)(recio_access_t *access,
      recio_stream_t *stream, int len, unsigned char *buf, char *key_string);
```

For random process and/or for record identification the supplied keys are expected to be in a non-display internal form. For the purposes of reporting and showing record addresses and keys in a display form, the access may provide a set of transfer functions which convert a supplied key/address of a record into a display form and back into the internal form. In the absence of such a function, the recio library will supply a set of methods which simply convert the string into the character hexadecimal representation of the key or record address. If the conversion is successful, then the length of the key is returned, otherwise `-1` is returned and a message is placed in the `last_error` structure of the supplied stream.

6 Sample Access Method Module

```
/* File: sampleam.c
 *
 * This file contains the Code Magus Limited Standard Access Method. This
 * access method reads and writes records of a file of either fixed or
 * variable length records. For variable length records the RDW I/O library
 * is used to read and write the file.
 *
 * Author: Stephen R. Donaldson [www.codemagus.com].
 *
```

```
   * Copyright (c) 2008 Code Magus Limited. All rights reserved.
   */

  /*
   * $Author: hayward $
   * $Date: 2009/02/06 15:18:40 $
   * $Id: sampleam.c,v 1.4 2009/02/06 15:18:40 hayward Exp $
   * $Name:  $
   * $Revision: 1.4 $
   * $State: Exp $
   *
   * $Log: sampleam.c,v $
   * Revision 1.4  2009/02/06 15:18:40  hayward
   * Change contact details.
   *
   * Revision 1.3  2008/03/27 18:45:52  stephen
   * Windows changes
   *
   * Revision 1.2  2008/03/11 15:48:32  hayward
   * Fix function names and recfm checking.
   *
   * Revision 1.1  2008/03/10 17:45:47  stephen
   * Change sample access method from standard
   *
   * Revision 1.5  2008/02/14 19:23:20  stephen
   * Add support for AMD supported modes and checks and skipseq mode
   *
   * Revision 1.4  2008/02/14 17:44:44  stephen
   * Changes and checks for skip sequential processing
   *
   * Revision 1.3  2008/02/08 10:22:46  stephen
   * Changes to support point and tell in sampleam access method
   *
   * Revision 1.2  2008/02/07 09:47:30  stephen
   * Fix problems encountered with testprog.c
   *
   * Revision 1.1  2008/02/07 06:12:35  stephen
   * Add all new files to module
   *
   */

static char *cvs_sampleam_c =
   "$Id: sampleam.c,v 1.4 2009/02/06 15:18:40 hayward Exp $";


  /*
   * Large file support required for various environments:
   */

#define _LARGEFILE_SOURCE
#define _LARGEFILE64_SOURCE
```

```
#define _LARGE_FILES
#define _FILE_OFFSET_BITS 64


#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#include <reciol.h>
#include <rdwio.h>

  /*
   * Types and structures:
   */

typedef struct stream_data stream_data_t;

  /*
   * The stream_data structure address will be kept in the stream provate
   * data field (stream_data). This is where all the underlying file info
   * will be kept.
   */

struct stream_data
   {
   FILE *file;         /* underlying file that contains the record stream */
   int userdw;         /* indicates whether the RDW I/O library is used */
   int reclen;         /* record length when refm = f */
   char *object;       /* name of file to which I/O occurs */
   fpos_t pos;         /* position of last record read/written/point */
   };

  /*
   * Exported functions:
   */

int sampleam_init(recio_access_t *access, unsigned long flags);

  /*
   * Local prototypes:
   */

static int sam_open(recio_access_t *access, recio_stream_t *stream,
      char *object, recio_modes_t mode, unsigned long flags);
static int sam_close(recio_access_t *access, recio_stream_t *stream);
static int sam_read(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf);
static int sam_write(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf);
static int sam_point(recio_access_t *access, recio_stream_t *stream,
```

```
      int len, unsigned char *buf);
static int sam_tell(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf);
static int sam_key2string(recio_access_t *access, recio_stream_t *stream,
            int len, unsigned char *buf, char *key_string);
static int sam_string2key(recio_access_t *access, recio_stream_t *stream,
            int len, unsigned char *buf, char *key_string);


  /* Function sampleam_init() is called to complete the initialisation of the
   * access method structure and is called the first time an object using the
   * access method is opened. The function indicates that the initialisation
   * completed successfully by returning a 0; otherwise the function should
   * insert a message into the last_error field of the structure and return
   * a -1.
   */

int sampleam_init(recio_access_t *access, unsigned long flags)
   {
   access->open = sam_open;
   access->close = sam_close;
   access->read = sam_read;
   access->write = sam_write;
   access->point = sam_point;
   access->tell = sam_tell;
   /*
   access->key2string = sam_key2string;
   access->string2key = sam_string2key;
   */

   return 0;
   } /* sampleam_init */


 /* Function sam_open() completes the open of a stream by opening the file
  * to which the records will be read/written.
  */

static int sam_open(recio_access_t *access, recio_stream_t *stream,
      char *object, recio_modes_t mode, unsigned long flags)
   {
   stream_data_t *stream_data;   /* underlying file operations data */
   char *open_mode;              /* open mode string */
   char *recfm;                  /* record format parameter */
   char *reclen;                 /* record length for fixed length records */
   int rc;

   stream_data = malloc(sizeof(*stream_data));
   memset(stream_data,0,sizeof(*stream_data));

   open_mode = access->option_lookup(stream,"mode");
   recfm = access->option_lookup(stream,"recfm");
```

```
   if (recfm[0] == 'v') stream_data->userdw = 1;
   else stream_data->userdw = 0;

   if (recfm[0] == 'f')
      {
      reclen = access->option_lookup(stream,"reclen");
      sscanf(reclen,"%d",&stream_data->reclen);
      }
   else
      stream_data->reclen = 0;

   if ((open_mode[0] == 'r' && mode != RECIO_MODE_SEQ_INPUT
            && mode != RECIO_MODE_SKIP_INPUT)
         || (open_mode[0] == 'w' && mode != RECIO_MODE_SEQ_OUTPUT))
      {
      sprintf(stream->last_error,
            "Open mode of %s conflicts with file open mode string %s.",
            object,open_mode);
      free(stream_data);
      return -1;
      }

   stream_data->object = object;
   stream_data->file = fopen(object,open_mode);
   if (!stream_data->file)
      {
      sprintf(stream->last_error,
            "Open of file %s failed: %s.",object,strerror(errno));
      free(stream_data);
      return -1;
      }

  /* The position of the file needs to be set once when the file is opened so
   * tha the position of the begining of the file is available before any
   * other operation changes the position of the file.
   */
   rc = fgetpos(stream_data->file,&stream_data->pos);
   if (rc)
      {
      sprintf(stream->last_error,
            "Error determining position in file %s (fgetpos error): %s.",
            stream_data->object,strerror(errno));
      return -1;
      }

   stream->stream_data = stream_data;
   return 0;
   } /* sam_open */

  /* Function sam_close() closes the file underlying the stream and frees
   * all access method related storage. If the operation succeeds, then
   * the function must return 0, otherwise a message must be formatted into
```

```
   * the last_error field of the stream structure and the function must
   * return -1.
   */

static int sam_close(recio_access_t *access, recio_stream_t *stream)
   {
   stream_data_t *stream_data;   /* underlying file operations data */

   stream_data = stream->stream_data;
   fclose(stream_data->file);
   free(stream_data);

   return 0;
   } /* sam_close */

  /* Function sam_read() reads the next record in a sequential file. If the
   * read operation fails then a message is formatted in the last_error
   * field of the stream. If the end of the file is found then the eof
   * status is set in the stream.
   */

static int sam_read(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf)
   {
   stream_data_t *stream_data;   /* underlying file operations data */
   int rc;

   stream_data = stream->stream_data;

  /* Before the read operation changes the position of the file, the position
   * needs to be extracted in case it is required by a subsequent recio_tell
   * operation.
   */
   rc = fgetpos(stream_data->file,&stream_data->pos);
   if (rc)
      {
      sprintf(stream->last_error,
            "Error determining position in file %s (fgetpos error): %s.",
            stream_data->object,strerror(errno));
      return -1;
      }

  /* Select the method of reading the file based the file being a variable
   * or fixed record file.
   */
   if (stream_data->userdw)
      rc = rdwio_fread(buf,len,stream_data->file);
   else
      rc = fread(buf,1,stream_data->reclen,stream_data->file);

  /* Report any I/O errors that have occured during the read operation.
   */
```

```
   if (rc < 0)
      {
      sprintf(stream->last_error,
            "I/O Error on read operation on %s: %s.",
            stream_data->object,strerror(errno));
      return -1;
      }

  /* Check for and report the end of file condition if it has occurred.
   */
   if (rc == 0)
      {
      access->set_eof(stream);
      return -1;
      }

  /* Check for and report a short read of data:
   */
   if (!stream_data->userdw && rc != stream_data->reclen)
      {
      sprintf(stream->last_error,
            "Invalid number of bytes returned from read. Expected %d and "
            "received %d.",stream_data->reclen,rc);
      return -1;
      }

  /* Otherwise the operation succeed.
   */
   return rc;
   } /* sam_read */

  /* Function sam_write() places the next logical record onto the output
   * stream. If the operation succeeds, then the function returns the
   * length of the data written. Otherwise the function returns -1 and
   * a message is formatted in the last_error field of the stream structure.
   */

static int sam_write(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf)
   {
   stream_data_t *stream_data;   /* underlying file operations data */
   int rc;

   stream_data = stream->stream_data;

  /* If the record lengths are supposed to be fixed length records, then
   * make sure that the supplied record length matches the record length
   * supplied at the time that the stream was opened.
   */
   if (!stream_data->userdw && len != stream_data->reclen)
      {
      sprintf(stream->last_error,
```

```
         "For write operations on fixed files, the supplied data length "
         "must match the record length supplied at the time that the file "
         "was opened. Open record length %d, write record length %d for %s.",
         stream_data->reclen,len,stream_data->object);
      return -1;
      }

/* Before the write operation changes the position of the file, the position
 * needs to be extracted in case it is required by a subsequent recio_tell
 * operation.
 */
 rc = fgetpos(stream_data->file,&stream_data->pos);
 if (rc)
    {
    sprintf(stream->last_error,
          "Error determining position in file %s (fgetpos error): %s.",
          stream_data->object,strerror(errno));
    return -1;
    }

/* Select the method of reading the file based the file being a variable
 * or fixed record file.
 */
 if (stream_data->userdw)
    rc = rdwio_fwrite(buf,len,stream_data->file);
 else
    rc = fwrite(buf,1,stream_data->reclen,stream_data->file);

/* Report any I/O errors that have occured during the write operation.
 */
 if (rc < 0)
    {
    sprintf(stream->last_error,
          "I/O Error on write operation on %s: %s.",
          stream_data->object,strerror(errno));
    return -1;
    }

/* Check for and report a short write of data:
 */
 if (rc != len)
    {
    sprintf(stream->last_error,
          "Short write returned from write operation. Expected %d and "
          "only wrote %d to %s.",stream_data->reclen,rc,stream_data->object);
    return -1;
    }

/* Otherwise the operation succeed.
 */
 return rc;
 } /* sam_write */
```

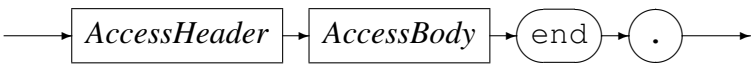
```
  /* Function sam_point() positions the file to the given file position. The
   * file position needs to be updated in the stream_data so that a subsequent
   * recio_tell operaion can determine the position.
   */

static int sam_point(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf)
   {
   stream_data_t *stream_data;   /* underlying file operations data */
   int rc;

   stream_data = stream->stream_data;


  /* Make sure that the supplied key is viable. This is just a check on the
   * lentgh of the supplied key.
   */
   if (len != sizeof(stream_data->pos))
      {
      sprintf(stream->last_error,
            "Invalid file key length. Expected %d, but received %d.",
            sizeof(stream_data->pos),len);
      return -1;
      }


  /* Update the stream_data version of the stream position so that a
   * subsequent recio_tell can determine the new position of the record.
   */
   memcpy(&stream_data->pos,buf,len);

  /* Attempt to reposition the file using the supplied key.
   */
   rc = fsetpos(stream_data->file,&stream_data->pos);
   if (rc)
      {
      sprintf(stream->last_error,
            "Error setting position in file %s (fsetpos error): %s.",
            stream_data->object,strerror(errno));
      return -1;
      }

   return 0;
   } /* sam_point */

static int sam_tell(recio_access_t *access, recio_stream_t *stream,
      int len, unsigned char *buf)
   {
   stream_data_t *stream_data;   /* underlying file operations data */

   stream_data = stream->stream_data;
```

```
  /* Make sure that the buffer for the key is large enough to insert the
   * key into.
   */
   if (len < sizeof(stream_data->pos))
      {
      sprintf(stream->last_error,
            "Supplied length of key buffer too short. %d required, only "
            "received %d.",sizeof(stream_data->pos),len);
      return -1;
      }

  /* The position of the recio stream is the position of the file stream
   * before the last record operation. This is the position saved by the
   * last operation.
   */
   memcpy(buf,&stream_data->pos,sizeof(stream_data->pos));
   return sizeof(stream_data->pos);
   } /* sam_tell */

/* not implemented by access method:
static int sam_key2string(recio_access_t *access, recio_stream_t *stream,
            int len, unsigned char *buf, char *key_string);
static int sam_string2key(recio_access_t *access, recio_stream_t *stream,
            int len, unsigned char *buf, char *key_string);
*/
```

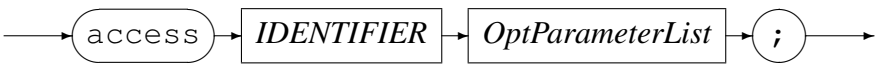
7 Access Method Definition Grammar

An *Access Method Definition* is a configuration file, referred to as an `amd` file which is also the file suffix or type, which defines an access method. An `amd` file has the following grammar:

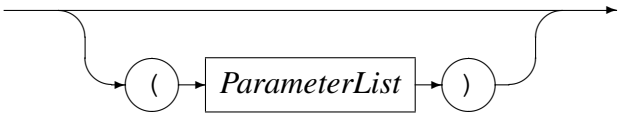
AccessSpecification



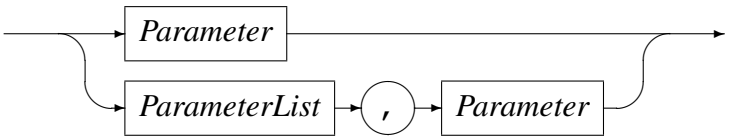
AccessHeader



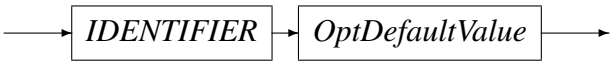
OptParameterList



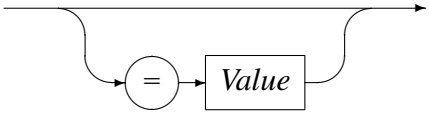
ParameterList



Parameter

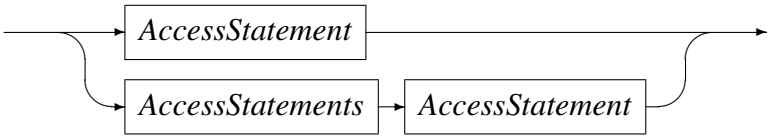


OptDefaultValue

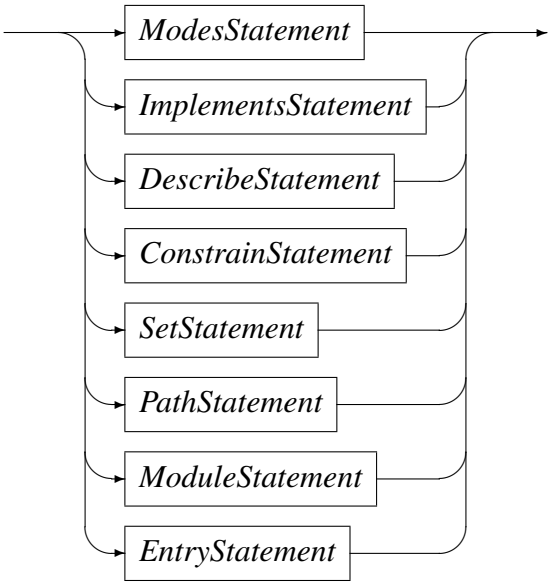


AccessBody

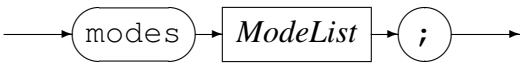
AccessStatements



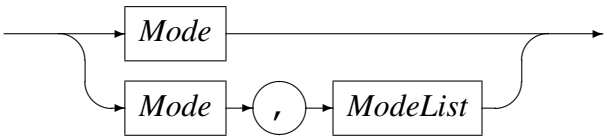
AccessStatement



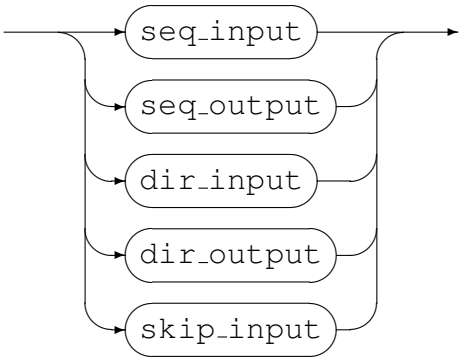
ModesStatement



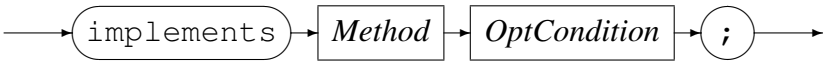
ModeList



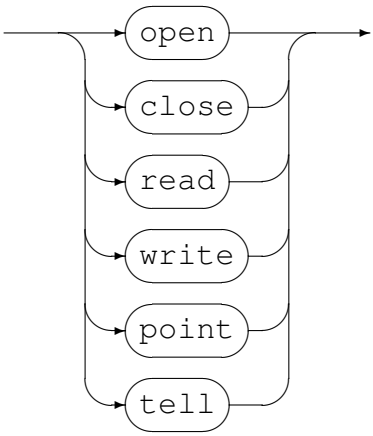
Mode



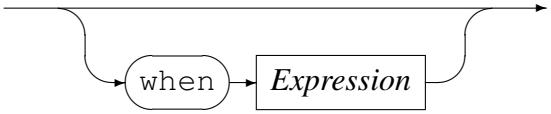
ImplementsStatement



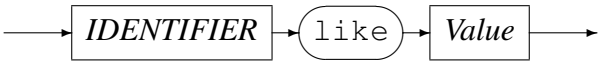
Method



OptCondition



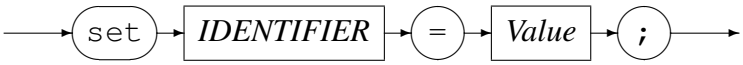
Expression



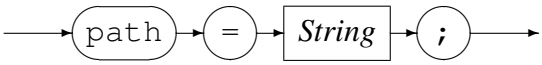
DescribeStatement

ConstrainStatement

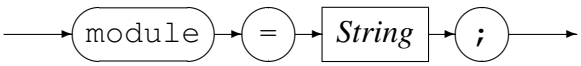
SetStatement



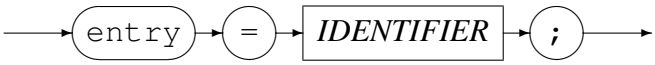
PathStatement



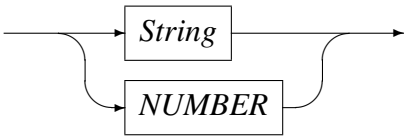
ModuleStatement



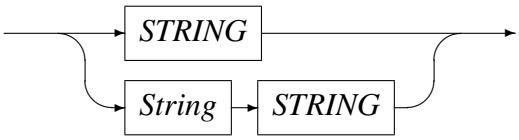
EntryStatement



Value



String



8 Sample Access Method Definition

The Access Method whose source module is shown as a sample in Section 6 on page 34 has the access method definition shown in this section. The access method definition file is expected to be called `SAMPLE.amd` (or equivalent, depending on the platform). Such an Access method would be nominated for usage by, for example, an open specification string of `sample(/tmp/in,recfm=v,mode=rb)`.

```
access sample(mode,recfm,reclen=32767);
```

```
  -- File: SAMPLE.amd
  --
  -- This file contains an access method definition which is used to read
  -- and write local files of fixed format records or rdw I/O library type
  -- records.

  -- Author: Stephen R. Donaldson [www.codemagus.com].


  -- Copyright (c) 2008 Code Magus Limited. All rights reserved.

   modes seq_input, seq_output, skip_input;

   implements open;
   implements close;
   implements read;
   implements write;
   implements point;
   implements tell;

   describe mode as
      "The mode is the open mode string which will be passed to the C Standard "
      "I/O Library.";

   describe recfm as
      "The record format parameter describes the format of the records "
      "expected to be in the file. The records are either fixed length "
      "records (in which case a record length must be provided), or they "
      "are variable in length and in which case the record length is "
      "ignored (if specified).";

   describe reclen as
      "For fixed format records a record length is required. This parameter "
      "supplies the record length in bytes.";

   constrain mode as "^[rw]b\(,type=record\)\?$";
   constrain recfm as "^[fv]$";
   constrain reclen as "^[1-9][0-9]*$";

   path = ${CODEMAGUS_AMDLIBS} "%s";
   module = "sampleam" ${CODEMAGUS_AMDSUFDL};
   entry = sampleam_init;

end.
```

References

[1] binary: Fixed and Variable Length Record Stream Access Method Version 1. CML Document CML00005-01, Code Magus Limited, July 2008. PDF.

[2] dataset: Catalog Access Method Definitions Version 1. CML Document CML00013-01, Code Magus Limited, July 2008. PDF.

[3] directory: Directory Record Stream Access Method Version 1. CML Document CML00014-01, Code Magus Limited, July 2008. PDF.

[4] MVS: MVS Record Stream Access Method Version 1. CML Document CML00016-01, Code Magus Limited, July 2008. PDF.

[5] objtypes: Configuring for Object Recognition, Generation and Manipulation. CML Document CML00018-01, Code Magus Limited, July 2008. PDF.

[6] RECIO: Thistle Type A Interface to the Code Magus Record Stream I/O Library. CML Document CML00021-01, Code Magus Limited, July 2008. PDF.

[7] remote: Remote Record Stream Access Method Version 1. CML Document CML00022-01, Code Magus Limited, July 2008. PDF.

[8] text: File Access Method Using POSIX Streams Version 1. CML Document CML00031-01, Code Magus Limited, July 2008. PDF.

[9] image: DB2 Image Copy Reader Access Method Version 1. CML Document CML00036-01, Code Magus Limited, July 2008. PDF.