ISO8211Lib

Contents

Chapter 1

ISO8211Lib

Introduction

ISO8211Lib is intended to be a simple reader for ISO/IEC 8211 formatted files, particularly those that are part of SDTS and S-57 datasets. It consists of open source, easy to compile and integrate C++ code.

ISO 8211 Background

The ISO 8211 FAQ has some good background on ISO 8211 formatted files. I will briefly introduce it here, with reference to the library classes representing the components.

An 8211 file (DDFModule) consists of a series of logical records. The fiarst record is special, and is called the DDR (Data Description Record). It basically contains definitions of all the data objects (fields or DDFFieldDefn objects) that can occur on the following data records.

The remainder of the records are known as DRs (data records - DDFRecord). They each contain one or more field (DDFField) instances. What fields appear on what records is not defined by ISO 8211, though more specific requirements may be implied by a particular data standard such as SDTS or S-57.

Each field instance has a name, and consists of a series of subfields. A given field always has the same subfields in each field instance, and these subfields are defined in the DDR (DDFSubfieldDefn), in association with their field definition (DDFFieldDefn). A field may appear 0, 1, or many times in a DR.

Each subfield has a name, format (from the DDFSubfieldDefn) and actual subfield data for a particular DR. Some fields contain an *array* of their group of subfields. For instance a *coordinate field* may have X and Y subfields, and they may repeat many times within one coordinate field indicating a series of points.

This would be a real good place for a UML diagram of ISO 8211, and the corresponding library classes!

Development Information

The **iso8211.h** contains the definitions for all public ISO8211Lib classes, enumerations and other services.

To establish access to an ISO 8211 dataset, instantiate a DDFModule object, and then use the DDFModule::Open() method. This will read the DDR, and establish all the DDFFieldDefn, and DDFSubfieldDefn objects which can be queried off the DDFModule.

The use DDFModule::ReadRecord() to fetch data records (DDFRecord). When a record is read, a list of field objects (DDFField) on that record are created. They can be queried with various DDFRecord methods.

Data pointers for individual subfields of a DDFField can be fetched with DDFField::GetSubfieldData().

The interpreted value can then be extracted with the appropriate one of DDFSubfieldDefn::ExtractIntValue(), DDFSubfieldDefn::ExtractStringValue(), or DDFSubfieldDefn::ExtractFloatValue(). Note that there is no object instantiated for individual subfields of a DDFField. Instead the application extracts a pointer to the subfields raw data, and then uses the DDFSubfieldDefn for that subfield to extract a useable value from the raw data.

Once the end of the file has been encountered (DDFModule::ReadRecord() returns NULL), the DDFModule should be deleted, which will close the file, and cleanup all records, definitions and related objects.

Class APIs

- DDFModule class.
- DDFFieldDefn class.
- DDFSubfieldDefn class.
- DDFRecord class.
- DDFField class.

A complete `Example Reader` should clarify simple use of ISO8211Lib.

Related Information

- The ISO 8211 standard can be ordered through `ISO`. It cost me about $200CDN.
- The `ISO/IEC 8211/DDFS Home Page` contains tutorials and some code by Dr. Alfred A. Brooks, one of the originators of the 8211 standard.
- The `ISO/IEC 8211 Home Page` has some python code for parsing 8211 files, and some other useful background.
- The `SDTS++` library from the USGS includes support for ISO 8211. It doesn't include some of the 1994 additions to ISO 8211, but it is sufficient for SDTS, and quite elegantly done. Also supports writing ISO 8211 files.
- The USGS also has an older `FIPS123` library which supports the older profile of ISO 8211 (to some extent).

Licensing

This library is offered as `Open Source`. In particular, it is offered under the X Consortium license which doesn't attempt to impose any copyleft, or credit requirements on users of the code.

The precise license text is:

Copyright (c) 1999, Frank Warmerdam

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Building the Source

1. First, fetch the source. The most recent source should be accessable at an url such as `http://home.gdal.org/projects/iso8211/iso8211lib-1.4.zip`.
2. Untar the source.

   ```
   % unzip iso8211lib-1.4.zip
   ```

3. On unix you can now type "configure" to establish configuration options.
4. On unix you can now type make to build libiso8211.a, and the sample mainline 8211view.

Windows developers will have to create their own makefile/project but can base it on the very simple Makefile.in provided. As well, you would need to copy cpl_config.h.in to cpl_config.h, and modify as needed. The default will likely work OK, but may result in some compiler warnings. Let me know if you are having difficulties, and I will prepare a VC++ makefile.

Author and Acknowledgements

The primary author of ISO8211Lib is `Frank Warmerdam`, and I can be reached at `warmerdam@pobox.com`. I am eager to receive bug reports, and also open to praise or suggestions.

I would like to thank:

- `Safe Software` who funded development of this library, and agreed for it to be Open Source.
- Mark Colletti, a primary author of SDTS++ from which I derived most of what I know about ISO 8211 and who was very supportive, answering a variety of questions.
- Tony J Ibbs, author of the ISO/IEC 8211 home page who answered a number of questions, and collected a variety of very useful information.
- Rodney Jenson, for a detailed bug report related to repeating variable length fields (from S-57).

I would also like to dedicate this library to the memory of Sol Katz. Sol released a variety of SDTS (and hence ISO8211) translators, at substantial personal effort, to the GIS community along with the many other generous contributions he made to the community. His example has been an inspiration to me, and I hope similar efforts on my part will contribute to his memory.

Chapter 2

ISO8211_Example

```
/* ******************************************************************************
 * $Id: 8211view.cpp 10645 2007-01-18 02:22:39Z warmerdam $
 *
 * Project:  SDTS Translator
 * Purpose:  Example program dumping data in 8211 data to stdout.
 * Author:   Frank Warmerdam, warmerdam@pobox.com
 *
 ******************************************************************************
 * Copyright (c) 1999, Frank Warmerdam
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 * DEALINGS IN THE SOFTWARE.
 ******************************************************************************/

#include <stdio.h>
#include "iso8211.h"

CPL_CVSID("$Id: 8211view.cpp 10645 2007-01-18 02:22:39Z warmerdam $");

static void ViewRecordField( DDFField * poField );
static int ViewSubfield( DDFSubfieldDefn *poSFDefn,
                         const char * pachFieldData,
                         int nBytesRemaining );

/* ******************************************************************/
/*                               main()                             */
/* ******************************************************************/

int main( int nArgc, char ** papszArgv )

{
    DDFModule   oModule;
    const char  *pszFilename = NULL;
    int         bFSPTHack = FALSE;

    for( int iArg = 1; iArg < nArgc; iArg++ )
    {
        if( EQUAL(papszArgv[iArg],"-fspt_repeating") )
            bFSPTHack = TRUE;
        else
            pszFilename = papszArgv[iArg];
    }

    if( pszFilename == NULL )
    {
        printf( "Usage: 8211view filename\n" );
        exit( 1 );
    }

/* -------------------------------------------------------------------- */
/*      Open the file.  Note that by default errors are reported to     */
/*      stderr, so we don't bother doing it ourselves.                  */
```

```
/* -------------------------------------------------------------------- */
    if( !oModule.Open( pszFilename ) )
    {
        exit( 1 );
    }

    if( bFSPTHack )
    {
        DDFFieldDefn *poFSPT = oModule.FindFieldDefn( "FSPT" );

        if( poFSPT == NULL )
            fprintf( stderr,
                     "unable to find FSPT field to set repeating flag.\n" );
        else
            poFSPT->SetRepeatingFlag( TRUE );
    }

/* -------------------------------------------------------------------- */
/*      Loop reading records till there are none left.                  */
/* -------------------------------------------------------------------- */
    DDFRecord   *poRecord;
    int         iRecord = 0;

    while( (poRecord = oModule.ReadRecord()) != NULL )
    {
        printf( "Record %d (%d bytes)\n",
                ++iRecord, poRecord->GetDataSize() );

        /* ---------------------------------------------------------- */
        /*      Loop over each field in this particular record.       */
        /* ---------------------------------------------------------- */
        for( int iField = 0; iField < poRecord->GetFieldCount(); iField++ )
        {
            DDFField    *poField = poRecord->GetField( iField );

            ViewRecordField( poField );
        }
    }
}

/* ********************************************************************/
/*                          ViewRecordField()                         */
/*                                                                    */
/*      Dump the contents of a field instance in a record.            */
/* ********************************************************************/

static void ViewRecordField( DDFField * poField )

{
    int         nBytesRemaining;
    const char  *pachFieldData;
    DDFFieldDefn *poFieldDefn = poField->GetFieldDefn();

    // Report general information about the field.
    printf( "    Field %s: %s\n",
            poFieldDefn->GetName(), poFieldDefn->GetDescription() );

    // Get pointer to this fields raw data.  We will move through
    // it consuming data as we report subfield values.

    pachFieldData = poField->GetData();
    nBytesRemaining = poField->GetDataSize();

    /* ------------------------------------------------------- */
    /*      Loop over the repeat count for this fields         */
    /*      subfields.  The repeat count will almost           */
    /*      always be one.                                     */
```

```
    /* -------------------------------------------------------- */
    int         iRepeat;

    for( iRepeat = 0; iRepeat < poField->GetRepeatCount(); iRepeat++ )
    {

        /* -------------------------------------------------------- */
        /*   Loop over all the subfields of this field, advancing   */
        /*   the data pointer as we consume data.                   */
        /* -------------------------------------------------------- */
        int     iSF;

        for( iSF = 0; iSF < poFieldDefn->GetSubfieldCount(); iSF++ )
        {
            DDFSubfieldDefn *poSFDefn = poFieldDefn->GetSubfield( iSF );
            int         nBytesConsumed;

            nBytesConsumed = ViewSubfield( poSFDefn, pachFieldData,
                                           nBytesRemaining );

            nBytesRemaining -= nBytesConsumed;
            pachFieldData += nBytesConsumed;
        }
    }
}

/* ************************************************************************/
/*                           ViewSubfield()                               */
/* ************************************************************************/

static int ViewSubfield( DDFSubfieldDefn *poSFDefn,
                         const char * pachFieldData,
                         int nBytesRemaining )

{
    int         nBytesConsumed = 0;

    switch( poSFDefn->GetType() )
    {
      case DDFInt:
        if( poSFDefn->GetBinaryFormat() == DDFSubfieldDefn::UInt )
            printf( "        %s = %u\n",
                    poSFDefn->GetName(),
                    poSFDefn->ExtractIntData( pachFieldData, nBytesRemaining,
                                              &nBytesConsumed ) );
        else
            printf( "        %s = %d\n",
                    poSFDefn->GetName(),
                    poSFDefn->ExtractIntData( pachFieldData, nBytesRemaining,
                                              &nBytesConsumed ) );
        break;

      case DDFFloat:
        printf( "        %s = %f\n",
                poSFDefn->GetName(),
                poSFDefn->ExtractFloatData( pachFieldData, nBytesRemaining,
                                            &nBytesConsumed ) );
        break;

      case DDFString:
        printf( "        %s = '%s'\n",
                poSFDefn->GetName(),
                poSFDefn->ExtractStringData( pachFieldData, nBytesRemaining,
                                             &nBytesConsumed ) );
        break;

      case DDFBinaryString:
```

```
      {
          int   i;
          //rjensen 19-Feb-2002 5 integer variables to decode NAME and LNAM
          int vrid_rcnm=0;
          int vrid_rcid=0;
          int foid_agen=0;
          int foid_find=0;
          int foid_fids=0;

          GByte *pabyBString = (GByte *)
              poSFDefn->ExtractStringData( pachFieldData, nBytesRemaining,
                                           &nBytesConsumed );

          printf( "        %s = 0x", poSFDefn->GetName() );
          for( i = 0; i < MIN(nBytesConsumed,24); i++ )
              printf( "%02X", pabyBString[i] );

          if( nBytesConsumed > 24 )
              printf( "%s", "..." );

          // rjensen 19-Feb-2002 S57 quick hack. decode NAME and LNAM bitfields
          if ( EQUAL(poSFDefn->GetName(),"NAME") )
          {
              vrid_rcnm=pabyBString[0];
              vrid_rcid=pabyBString[1] + (pabyBString[2]*256)+
                  (pabyBString[3]*65536)+ (pabyBString[4]*16777216);
              printf("\tVRID RCNM = %d,RCID = %u",vrid_rcnm,vrid_rcid);
          }
          else if ( EQUAL(poSFDefn->GetName(),"LNAM") )
          {
              foid_agen=pabyBString[0] + (pabyBString[1]*256);
              foid_find=pabyBString[2] + (pabyBString[3]*256)+
                  (pabyBString[4]*65536)+ (pabyBString[5]*16777216);
              foid_fids=pabyBString[6] + (pabyBString[7]*256);
              printf("\tFOID AGEN = %u,FIDN = %u,FIDS = %u",
                     foid_agen,foid_find,foid_fids);
          }

          printf( "\n" );
      }
      break;

    }

    return nBytesConsumed;
}
```

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

DDFField . . . . . ??
DDFFieldDefn . . . . . ??
DDFModule . . . . . ??
DDFRecord . . . . . ??
DDFSubfieldDefn . . . . . ??

Chapter 4

Class Documentation

4.1 DDFField Class Reference

`#include <iso8211.h>`

Public Member Functions

- void Dump (FILE *fp)
- const char * GetSubfieldData (DDFSubfieldDefn *, int *=NULL, int=0)
- const char * GetInstanceData (int nInstance, int *pnSize)
- const char * GetData ()
- int GetDataSize ()
- int GetRepeatCount ()
- DDFFieldDefn * GetFieldDefn ()

4.1.1 Detailed Description

This object represents one field in a DDFRecord. This models an instance of the fields data, rather than it's data definition which is handled by the DDFFieldDefn class. Note that a DDFField doesn't have DDFSubfield children as you would expect. To extract subfield values use GetSubfieldData() to find the right data pointer and then use ExtractIntData(), ExtractFloatData() or ExtractStringData().

4.1.2 Member Function Documentation

4.1.2.1 void DDFField::Dump (FILE * *fp*)

Write out field contents to debugging file.

A variety of information about this field, and all it's subfields is written to the given debugging file handle. Note that field definition information (ala DDFFieldDefn) isn't written.

Parameters:

fp The standard io file handle to write to. ie. stderr

4.1.2.2 const char * DDFField::GetSubfieldData (DDFSubfieldDefn * *poSFDefn*, int * *pnMaxBytes* = NULL, int *iSubfieldIndex* = 0)

Fetch raw data pointer for a particular subfield of this field.

The passed DDFSubfieldDefn (poSFDefn) should be acquired from the DDFFieldDefn corresponding with this field. This is normally done once before reading any records. This method involves a series of calls to DDFSubfield::GetDataLength() in order to track through the DDFField data to that belonging to the requested subfield. This can be relatively expensive.

Parameters:

poSFDefn The definition of the subfield for which the raw data pointer is desired.

pnMaxBytes The maximum number of bytes that can be accessed from the returned data pointer is placed in this int, unless it is NULL.

iSubfieldIndex The instance of this subfield to fetch. Use zero (the default) for the first instance.

Returns:

A pointer into the DDFField's data that belongs to the subfield. This returned pointer is invalidated by the next record read (DDFRecord::ReadRecord()) and the returned pointer should not be freed by the application.

4.1.2.3 const char * DDFField::GetInstanceData (int *nInstance*, int * *pnInstanceSize*)

Get field instance data and size.

The returned data pointer and size values are suitable for use with DDFRecord::SetFieldRaw().

Parameters:

nInstance a value from 0 to GetRepeatCount()-1.

pnInstanceSize a location to put the size (in bytes) of the field instance data returned. This size will include the unit terminator (if any), but not the field terminator. This size pointer may be NULL if not needed.

Returns:

the data pointer, or NULL on error.

4.1.2.4 const char* DDFField::GetData () [inline]

Return the pointer to the entire data block for this record. This is an internal copy, and shouldn't be freed by the application.

4.1.2.5 int DDFField::GetDataSize () [inline]

Return the number of bytes in the data block returned by GetData().

4.1.2.6 int DDFField::GetRepeatCount ()

How many times do the subfields of this record repeat? This will always be one for non-repeating fields.

Returns:

The number of times that the subfields of this record occur in this record. This will be one for non-repeating fields.

See also:

8211view example program for demonstation of handling repeated fields properly.

4.1.2.7 DDFFieldDefn* DDFField::GetFieldDefn () [inline]

Fetch the corresponding DDFFieldDefn.

The documentation for this class was generated from the following files:

- iso8211.h
- ddffield.cpp

4.2 DDFFieldDefn Class Reference

`#include <iso8211.h>`

Public Member Functions

- void Dump (FILE *fp)
- const char * GetName ()
- const char * GetDescription ()
- int GetSubfieldCount ()
- DDFSubfieldDefn * GetSubfield (int i)
- DDFSubfieldDefn * FindSubfieldDefn (const char *)
- int GetFixedWidth ()
- int IsRepeating ()
- void SetRepeatingFlag (int n)
- char * GetDefaultValue (int *pnSize)

4.2.1 Detailed Description

Information from the DDR defining one field. Note that just because a field is defined for a DDFModule doesn't mean that it actually occurs on any records in the module. DDFFieldDefns are normally just significant as containers of the DDFSubfieldDefns.

4.2.2 Member Function Documentation

4.2.2.1 void DDFFieldDefn::Dump (FILE * *fp*)

Write out field definition info to debugging file.

A variety of information about this field definition, and all it's subfields is written to the give debugging file handle.

Parameters:

fp The standard io file handle to write to. ie. stderr

4.2.2.2 const char* DDFFieldDefn::GetName () `[inline]`

Fetch a pointer to the field name (tag).

Returns:

this is an internal copy and shouldn't be freed.

4.2.2.3 const char* DDFFieldDefn::GetDescription () `[inline]`

Fetch a longer descriptio of this field.

Returns:

this is an internal copy and shouldn't be freed.

4.2.2.4 int DDFFieldDefn::GetSubfieldCount () [inline]

Get the number of subfields.

4.2.2.5 DDFSubfieldDefn * DDFFieldDefn::GetSubfield (int *i*)

Fetch a subfield by index.

Parameters:

i The index subfield index. (Between 0 and GetSubfieldCount()-1)

Returns:

The subfield pointer, or NULL if the index is out of range.

4.2.2.6 DDFSubfieldDefn * DDFFieldDefn::FindSubfieldDefn (const char * *pszMnemonic*)

Find a subfield definition by it's mnemonic tag.

Parameters:

pszMnemonic The name of the field.

Returns:

The subfield pointer, or NULL if there isn't any such subfield.

4.2.2.7 int DDFFieldDefn::GetFixedWidth () [inline]

Get the width of this field. This function isn't normally used by applications.

Returns:

The width of the field in bytes, or zero if the field is not apparently of a fixed width.

4.2.2.8 int DDFFieldDefn::IsRepeating () [inline]

Fetch repeating flag.

See also:

DDFField::GetRepeatCount()

Returns:

TRUE if the field is marked as repeating.

4.2.2.9 void DDFFieldDefn::SetRepeatingFlag (int *n*) [inline]

this is just for an S-57 hack for swedish data

4.2.2.10 char * DDFFieldDefn::GetDefaultValue (int * *pnSize*)

Return default data for field instance.

The documentation for this class was generated from the following files:

- iso8211.h
- ddffielddefn.cpp

4.3 DDFModule Class Reference

`#include <iso8211.h>`

Public Member Functions

- DDFModule ()
- ~DDFModule ()
- int Open (const char *pszFilename, int bFailQuietly=FALSE)
- void Close ()
- void Dump (FILE *fp)
- DDFRecord * ReadRecord (void)
- void Rewind (long nOffset=-1)
- DDFFieldDefn * FindFieldDefn (const char *)
- int GetFieldCount ()
- DDFFieldDefn * GetField (int)
- void AddField (DDFFieldDefn *poNewFDefn)

4.3.1 Detailed Description

The primary class for reading ISO 8211 files. This class contains all the information read from the DDR record, and is used to read records from the file.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 DDFModule::DDFModule ()

The constructor.

4.3.2.2 DDFModule::~DDFModule ()

The destructor.

4.3.3 Member Function Documentation

4.3.3.1 int DDFModule::Open (const char * *pszFilename*, int *bFailQuietly* = `FALSE`)

Open a ISO 8211 (DDF) file for reading.

If the open succeeds the data descriptive record (DDR) will have been read, and all the field and subfield definitions will be available.

Parameters:

pszFilename The name of the file to open.

bFailQuietly If FALSE a CPL Error is issued for non-8211 files, otherwise quietly return NULL.

Returns:

FALSE if the open fails or TRUE if it succeeds. Errors messages are issued internally with CPLError().

4.3.3.2 void DDFModule::Close ()

Close an ISO 8211 file.

4.3.3.3 void DDFModule::Dump (FILE * *fp*)

Write out module info to debugging file.

A variety of information about the module is written to the debugging file. This includes all the field and subfield definitions read from the header.

Parameters:

fp The standard io file handle to write to. ie. stderr.

4.3.3.4 DDFRecord * DDFModule::ReadRecord (void)

Read one record from the file.

Returns:

A pointer to a DDFRecord object is returned, or NULL if a read error, or end of file occurs. The returned record is owned by the module, and should not be deleted by the application. The record is only valid untill the next ReadRecord() at which point it is overwritten.

4.3.3.5 void DDFModule::Rewind (long *nOffset* = -1)

Return to first record.

The next call to ReadRecord() will read the first data record in the file.

Parameters:

nOffset the offset in the file to return to. By default this is -1, a special value indicating that reading should return to the first data record. Otherwise it is an absolute byte offset in the file.

4.3.3.6 DDFFieldDefn * DDFModule::FindFieldDefn (const char * *pszFieldName*)

Fetch the definition of the named field.

This function will scan the DDFFieldDefn's on this module, to find one with the indicated field name.

Parameters:

pszFieldName The name of the field to search for. The comparison is case insensitive.

Returns:

A pointer to the request DDFFieldDefn object is returned, or NULL if none matching the name are found. The return object remains owned by the DDFModule, and should not be deleted by application code.

4.3.3.7 int DDFModule::GetFieldCount () [inline]

Fetch the number of defined fields.

4.3.3.8 DDFFieldDefn * DDFModule::GetField (int *i*)

Fetch a field definition by index.

Parameters:

i (from 0 to GetFieldCount() - 1.

Returns:

the returned field pointer or NULL if the index is out of range.

4.3.3.9 void DDFModule::AddField (DDFFieldDefn * *poNewFDefn*)

Add new field definition.

Field definitions may only be added to DDFModules being used for writing, not those being used for reading. Ownership of the DDFFieldDefn object is taken by the DDFModule.

Parameters:

poNewFDefn definition to be added to the module.

The documentation for this class was generated from the following files:

- iso8211.h
- ddfmodule.cpp

4.4 DDFRecord Class Reference

`#include <iso8211.h>`

Public Member Functions

- DDFRecord * Clone ()
- DDFRecord * CloneOn (DDFModule *)
- void Dump (FILE *)
- int GetFieldCount ()
- DDFField * FindField (const char *, int=0)
- DDFField * GetField (int)
- int GetIntSubfield (const char *, int, const char *, int, int *=NULL)
- double GetFloatSubfield (const char *, int, const char *, int, int *=NULL)
- const char * GetStringSubfield (const char *, int, const char *, int, int *=NULL)
- int SetIntSubfield (const char *pszField, int iFieldIndex, const char *pszSubfield, int iSubfieldIndex, int nValue)
- int SetStringSubfield (const char *pszField, int iFieldIndex, const char *pszSubfield, int iSubfieldIndex, const char *pszValue, int nValueLength=-1)
- int SetFloatSubfield (const char *pszField, int iFieldIndex, const char *pszSubfield, int iSubfieldIndex, double dfNewValue)
- int GetDataSize ()
- const char * GetData ()
- DDFModule * GetModule ()
- int ResizeField (DDFField *poField, int nNewDataSize)
- int DeleteField (DDFField *poField)
- DDFField * AddField (DDFFieldDefn *)
- int CreateDefaultFieldInstance (DDFField *poField, int iIndexWithinField)
- int SetFieldRaw (DDFField *poField, int iIndexWithinField, const char *pachRawData, int nRawDataSize)
- int Write ()

4.4.1 Detailed Description

Contains instance data from one data record (DR). The data is contained as a list of DDFField instances partitioning the raw data into fields.

4.4.2 Member Function Documentation

4.4.2.1 DDFRecord * DDFRecord::Clone ()

Make a copy of a record.

This method is used to make a copy of a record that will become (mostly) the properly of application. However, it is automatically destroyed if the DDFModule it was created relative to is destroyed, as it's field and subfield definitions relate to that DDFModule. However, it does persist even when the record returned by DDFModule::ReadRecord() is invalidated, such as when reading a new record. This allows an application to cache whole DDFRecords.

Returns:

A new copy of the DDFRecord. This can be delete'd by the application when no longer needed, otherwise it will be cleaned up when the DDFModule it relates to is destroyed or closed.

4.4.2.2 DDFRecord * DDFRecord::CloneOn (DDFModule * *poTargetModule*)

Recreate a record referencing another module.

Works similarly to the DDFRecord::Clone() method, but creates the new record with reference to a different DDFModule. All DDFFieldDefn references are transcribed onto the new module based on field names. If any fields don't have a similarly named field on the target module the operation will fail. No validation of field types and properties is done, but this operation is intended only to be used between modules with matching definitions of all affected fields.

The new record will be managed as a clone by the target module in a manner similar to regular clones.

Parameters:

poTargetModule the module on which the record copy should be created.

Returns:

NULL on failure or a pointer to the cloned record.

4.4.2.3 void DDFRecord::Dump (FILE * *fp*)

Write out record contents to debugging file.

A variety of information about this record, and all it's fields and subfields is written to the given debugging file handle. Note that field definition information (ala DDFFieldDefn) isn't written.

Parameters:

fp The standard io file handle to write to. ie. stderr

4.4.2.4 int DDFRecord::GetFieldCount () `[inline]`

Get the number of DDFFields on this record.

4.4.2.5 DDFField * DDFRecord::FindField (const char * *pszName*, int *iFieldIndex* = `0`)

Find the named field within this record.

Parameters:

pszName The name of the field to fetch. The comparison is case insensitive.

iFieldIndex The instance of this field to fetch. Use zero (the default) for the first instance.

Returns:

Pointer to the requested DDFField. This pointer is to an internal object, and should not be freed. It remains valid until the next record read.

4.4.2.6 DDFField * DDFRecord::GetField (int *i*)

Fetch field object based on index.

Parameters:

i The index of the field to fetch. Between 0 and GetFieldCount()-1.

Returns:

A DDFField pointer, or NULL if the index is out of range.

4.4.2.7 int DDFRecord::GetIntSubfield (const char * *pszField*, int *iFieldIndex*, const char * *pszSubfield*, int *iSubfieldIndex*, int * *pnSuccess* = NULL)

Fetch value of a subfield as an integer. This is a convenience function for fetching a subfield of a field within this record.

Parameters:

pszField The name of the field containing the subfield.

iFieldIndex The instance of this field within the record. Use zero for the first instance of this field.

pszSubfield The name of the subfield within the selected field.

iSubfieldIndex The instance of this subfield within the record. Use zero for the first instance.

pnSuccess Pointer to an int which will be set to TRUE if the fetch succeeds, or FALSE if it fails. Use NULL if you don't want to check success.

Returns:

The value of the subfield, or zero if it failed for some reason.

4.4.2.8 double DDFRecord::GetFloatSubfield (const char * *pszField*, int *iFieldIndex*, const char * *pszSubfield*, int *iSubfieldIndex*, int * *pnSuccess* = NULL)

Fetch value of a subfield as a float (double). This is a convenience function for fetching a subfield of a field within this record.

Parameters:

pszField The name of the field containing the subfield.

iFieldIndex The instance of this field within the record. Use zero for the first instance of this field.

pszSubfield The name of the subfield within the selected field.

iSubfieldIndex The instance of this subfield within the record. Use zero for the first instance.

pnSuccess Pointer to an int which will be set to TRUE if the fetch succeeds, or FALSE if it fails. Use NULL if you don't want to check success.

Returns:

The value of the subfield, or zero if it failed for some reason.

4.4.2.9 const char * DDFRecord::GetStringSubfield (const char * *pszField*, int *iFieldIndex*, const char * *pszSubfield*, int *iSubfieldIndex*, int * *pnSuccess* = NULL)

Fetch value of a subfield as a string. This is a convenience function for fetching a subfield of a field within this record.

Parameters:

pszField The name of the field containing the subfield.

iFieldIndex The instance of this field within the record. Use zero for the first instance of this field.

pszSubfield The name of the subfield within the selected field.

iSubfieldIndex The instance of this subfield within the record. Use zero for the first instance.

pnSuccess Pointer to an int which will be set to TRUE if the fetch succeeds, or FALSE if it fails. Use NULL if you don't want to check success.

Returns:

The value of the subfield, or NULL if it failed for some reason. The returned pointer is to internal data and should not be modified or freed by the application.

4.4.2.10 int DDFRecord::SetIntSubfield (const char * *pszField*, int *iFieldIndex*, const char * *pszSubfield*, int *iSubfieldIndex*, int *nNewValue*)

Set an integer subfield in record.

The value of a given subfield is replaced with a new integer value formatted appropriately.

Parameters:

pszField the field name to operate on.

iFieldIndex the field index to operate on (zero based).

pszSubfield the subfield name to operate on.

iSubfieldIndex the subfield index to operate on (zero based).

nNewValue the new value to place in the subfield.

Returns:

TRUE if successful, and FALSE if not.

4.4.2.11 int DDFRecord::SetStringSubfield (const char * *pszField*, int *iFieldIndex*, const char * *pszSubfield*, int *iSubfieldIndex*, const char * *pszValue*, int *nValueLength* = -1)

Set a string subfield in record.

The value of a given subfield is replaced with a new string value formatted appropriately.

Parameters:

pszField the field name to operate on.

iFieldIndex the field index to operate on (zero based).

pszSubfield the subfield name to operate on.

iSubfieldIndex the subfield index to operate on (zero based).

pszValue the new string to place in the subfield. This may be arbitrary binary bytes if nValueLength is specified.

nValueLength the number of valid bytes in pszValue, may be -1 to internally fetch with strlen().

Returns:

TRUE if successful, and FALSE if not.

4.4.2.12 int DDFRecord::SetFloatSubfield (const char * *pszField*, int *iFieldIndex*, const char * *pszSubfield*, int *iSubfieldIndex*, double *dfNewValue*)

Set a float subfield in record.

The value of a given subfield is replaced with a new float value formatted appropriately.

Parameters:

pszField the field name to operate on.

iFieldIndex the field index to operate on (zero based).

pszSubfield the subfield name to operate on.

iSubfieldIndex the subfield index to operate on (zero based).

dfNewValue the new value to place in the subfield.

Returns:

TRUE if successful, and FALSE if not.

4.4.2.13 int DDFRecord::GetDataSize () [inline]

Fetch size of records raw data (GetData()) in bytes.

4.4.2.14 const char* DDFRecord::GetData () [inline]

Fetch the raw data for this record. The returned pointer is effectively to the data for the first field of the record, and is of size GetDataSize().

4.4.2.15 DDFModule* DDFRecord::GetModule () [inline]

Fetch the DDFModule with which this record is associated.

4.4.2.16 int DDFRecord::ResizeField (DDFField * *poField*, int *nNewDataSize*)

Alter field data size within record.

This method will rearrange a DDFRecord altering the amount of space reserved for one of the existing fields. All following fields will be shifted accordingly. This includes updating the DDFField infos, and actually moving stuff within the data array after reallocating to the desired size.

Parameters:

poField the field to alter.

nNewDataSize the number of data bytes to be reserved for the field.

Returns:

TRUE on success or FALSE on failure.

4.4.2.17 int DDFRecord::DeleteField (DDFField * *poTarget*)

Delete a field instance from a record.

Remove a field from this record, cleaning up the data portion and repacking the fields list. We don't try to reallocate the data area of the record to be smaller.

NOTE: This method doesn't actually remove the header information for this field from the record tag list yet. This should be added if the resulting record is even to be written back to disk!

Parameters:

poTarget the field instance on this record to delete.

Returns:

TRUE on success, or FALSE on failure. Failure can occur if poTarget isn't really a field on this record.

4.4.2.18 DDFField * DDFRecord::AddField (DDFFieldDefn * *poDefn*)

Add a new field to record.

Add a new zero sized field to the record. The new field is always added at the end of the record.

NOTE: This method doesn't currently update the header information for the record to include the field information for this field, so the resulting record image isn't suitable for writing to disk. However, everything else about the record state should be updated properly to reflect the new field.

Parameters:

poDefn the definition of the field to be added.

Returns:

the field object on success, or NULL on failure.

4.4.2.19 int DDFRecord::CreateDefaultFieldInstance (DDFField * *poField*, int *iIndexWithinField*)

Initialize default instance.

This method is normally only used internally by the AddField() method to initialize the new field instance with default subfield values. It installs default data for one instance of the field in the record using the DDFFieldDefn::GetDefaultValue() method and DDFRecord::SetFieldRaw().

Parameters:

poField the field within the record to be assign a default instance.

iIndexWithinField the instance to set (may not have been tested with values other than 0).

Returns:

TRUE on success or FALSE on failure.

4.4.2.20 int DDFRecord::SetFieldRaw (DDFField * *poField*, int *iIndexWithinField*, const char * *pachRawData*, int *nRawDataSize*)

Set the raw contents of a field instance.

Parameters:

poField the field to set data within.

iIndexWithinField The instance of this field to replace. Must be a value between 0 and GetRepeatCount(). If GetRepeatCount() is used, a new instance of the field is appeneded.

pachRawData the raw data to replace this field instance with.

nRawDataSize the number of bytes pointed to by pachRawData.

Returns:

TRUE on success or FALSE on failure.

4.4.2.21 int DDFRecord::Write ()

Write record out to module.

This method writes the current record to the module to which it is attached. Normally this would be at the end of the file, and only used for modules newly created with DDFModule::Create(). Rewriting existing records is not supported at this time. Calling Write() multiple times on a DDFRecord will result it multiple copies being written at the end of the module.

Returns:

TRUE on success or FALSE on failure.

The documentation for this class was generated from the following files:

- iso8211.h
- ddfrecord.cpp

4.5 DDFSubfieldDefn Class Reference

`#include <iso8211.h>`

Public Types

- enum DDFBinaryFormat

Public Member Functions

- const char * GetName ()
- const char * GetFormat ()
- DDFDataType GetType ()
- double ExtractFloatData (const char *pachData, int nMaxBytes, int *pnConsumedBytes)
- int ExtractIntData (const char *pachData, int nMaxBytes, int *pnConsumedBytes)
- const char * ExtractStringData (const char *pachData, int nMaxBytes, int *pnConsumedBytes)
- int GetDataLength (const char *, int, int *)
- void DumpData (const char *pachData, int nMaxBytes, FILE *fp)
- int FormatStringValue (char *pachData, int nBytesAvailable, int *pnBytesUsed, const char *pszValue, int nValueLength=-1)
- int FormatIntValue (char *pachData, int nBytesAvailable, int *pnBytesUsed, int nNewValue)
- int FormatFloatValue (char *pachData, int nBytesAvailable, int *pnBytesUsed, double dfNewValue)
- int GetWidth ()
- int GetDefaultValue (char *pachData, int nBytesAvailable, int *pnBytesUsed)
- void Dump (FILE *fp)

4.5.1 Detailed Description

Information from the DDR record describing one subfield of a DDFFieldDefn. All subfields of a field will occur in each occurance of that field (as a DDFField) in a DDFRecord. Subfield's actually contain formatted data (as instances within a record).

4.5.2 Member Enumeration Documentation

4.5.2.1 enum DDFSubfieldDefn::DDFBinaryFormat

Binary format: this is the digit immediately following the B or b for binary formats.

4.5.3 Member Function Documentation

4.5.3.1 const char* DDFSubfieldDefn::GetName () [inline]

Get pointer to subfield name.

4.5.3.2 const char* DDFSubfieldDefn::GetFormat () [inline]

Get pointer to subfield format string

4.5.3.3 DDFDataType DDFSubfieldDefn::GetType () [inline]

Get the general type of the subfield. This can be used to determine which of ExtractFloatData(), ExtractIntData() or ExtractStringData() should be used.

Returns:

The subfield type. One of DDFInt, DDFFloat, DDFString or DDFBinaryString.

4.5.3.4 double DDFSubfieldDefn::ExtractFloatData (const char * *pachSourceData*, int *nMaxBytes*, int * *pnConsumedBytes*)

Extract a subfield value as a float. Given a pointer to the data for this subfield (from within a DDFRecord) this method will return the floating point data for this subfield. The number of bytes consumed as part of this field can also be fetched. This method may be called for any type of subfield, and will return zero if the subfield is not numeric.

Parameters:

pachSourceData The pointer to the raw data for this field. This may have come from DDFRecord::GetData(), taking into account skip factors over previous subfields data.

nMaxBytes The maximum number of bytes that are accessable after pachSourceData.

pnConsumedBytes Pointer to an integer into which the number of bytes consumed by this field should be written. May be NULL to ignore. This is used as a skip factor to increment pachSourceData to point to the next subfields data.

Returns:

The subfield's numeric value (or zero if it isn't numeric).

See also:

ExtractIntData(), ExtractStringData()

4.5.3.5 int DDFSubfieldDefn::ExtractIntData (const char * *pachSourceData*, int *nMaxBytes*, int * *pnConsumedBytes*)

Extract a subfield value as an integer. Given a pointer to the data for this subfield (from within a DDFRecord) this method will return the int data for this subfield. The number of bytes consumed as part of this field can also be fetched. This method may be called for any type of subfield, and will return zero if the subfield is not numeric.

Parameters:

pachSourceData The pointer to the raw data for this field. This may have come from DDFRecord::GetData(), taking into account skip factors over previous subfields data.

nMaxBytes The maximum number of bytes that are accessable after pachSourceData.

pnConsumedBytes Pointer to an integer into which the number of bytes consumed by this field should be written. May be NULL to ignore. This is used as a skip factor to increment pachSourceData to point to the next subfields data.

Returns:

The subfield's numeric value (or zero if it isn't numeric).

See also:

ExtractFloatData(), ExtractStringData()

4.5.3.6 const char * DDFSubfieldDefn::ExtractStringData (const char * *pachSourceData*, int *nMaxBytes*, int * *pnConsumedBytes*)

Extract a zero terminated string containing the data for this subfield. Given a pointer to the data for this subfield (from within a DDFRecord) this method will return the data for this subfield. The number of bytes consumed as part of this field can also be fetched. This number may be one longer than the string length if there is a terminator character used.

This function will return the raw binary data of a subfield for types other than DDFString, including data past zero chars. This is the standard way of extracting DDFBinaryString subfields for instance.

Parameters:

pachSourceData The pointer to the raw data for this field. This may have come from DDFRecord::GetData(), taking into account skip factors over previous subfields data.

nMaxBytes The maximum number of bytes that are accessable after pachSourceData.

pnConsumedBytes Pointer to an integer into which the number of bytes consumed by this field should be written. May be NULL to ignore. This is used as a skip factor to increment pachSourceData to point to the next subfields data.

Returns:

A pointer to a buffer containing the data for this field. The returned pointer is to an internal buffer which is invalidated on the next ExtractStringData() call on this DDFSubfieldDefn(). It should not be freed by the application.

See also:

ExtractIntData(), ExtractFloatData()

4.5.3.7 int DDFSubfieldDefn::GetDataLength (const char * *pachSourceData*, int *nMaxBytes*, int * *pnConsumedBytes*)

Scan for the end of variable length data. Given a pointer to the data for this subfield (from within a DDFRecord) this method will return the number of bytes which are data for this subfield. The number of bytes consumed as part of this field can also be fetched. This number may be one longer than the length if there is a terminator character used.

This method is mainly for internal use, or for applications which want the raw binary data to interpret themselves. Otherwise use one of ExtractStringData(), ExtractIntData() or ExtractFloatData().

Parameters:

pachSourceData The pointer to the raw data for this field. This may have come from DDFRecord::GetData(), taking into account skip factors over previous subfields data.

nMaxBytes The maximum number of bytes that are accessable after pachSourceData.

pnConsumedBytes Pointer to an integer into which the number of bytes consumed by this field should be written. May be NULL to ignore.

Returns:

The number of bytes at pachSourceData which are actual data for this record (not including unit, or field terminator).

4.5.3.8 void DDFSubfieldDefn::DumpData (const char * *pachData*, int *nMaxBytes*, FILE * *fp*)

Dump subfield value to debugging file.

Parameters:

pachData Pointer to data for this subfield.

nMaxBytes Maximum number of bytes available in pachData.

fp File to write report to.

4.5.3.9 int DDFSubfieldDefn::FormatStringValue (char * *pachData*, int *nBytesAvailable*, int * *pnBytesUsed*, const char * *pszValue*, int *nValueLength* = -1)

Format string subfield value.

Returns a buffer with the passed in string value reformatted in a way suitable for storage in a DDFField for this subfield.

4.5.3.10 int DDFSubfieldDefn::FormatIntValue (char * *pachData*, int *nBytesAvailable*, int * *pnBytesUsed*, int *nNewValue*)

Format int subfield value.

Returns a buffer with the passed in int value reformatted in a way suitable for storage in a DDFField for this subfield.

4.5.3.11 int DDFSubfieldDefn::FormatFloatValue (char * *pachData*, int *nBytesAvailable*, int * *pnBytesUsed*, double *dfNewValue*)

Format float subfield value.

Returns a buffer with the passed in float value reformatted in a way suitable for storage in a DDFField for this subfield.

4.5.3.12 int DDFSubfieldDefn::GetWidth () [inline]

Get the subfield width (zero for variable).

4.5.3.13 int DDFSubfieldDefn::GetDefaultValue (char * *pachData*, int *nBytesAvailable*, int * *pnBytesUsed*)

Get default data.

Returns the default subfield data contents for this subfield definition. For variable length numbers this will normally be "0<unit-terminator>". For variable length strings it will be "<unit-terminator>". For fixed length numbers it is zero filled. For fixed length strings it is space filled. For binary numbers it is binary zero filled.

Parameters:

pachData the buffer into which the returned default will be placed. May be NULL if just querying default size.

nBytesAvailable the size of pachData in bytes.

pnBytesUsed will receive the size of the subfield default data in bytes.

Returns:

TRUE on success or FALSE on failure or if the passed buffer is too small to hold the default.

4.5.3.14 void DDFSubfieldDefn::Dump (FILE * *fp*)

Write out subfield definition info to debugging file.

A variety of information about this field definition is written to the give debugging file handle.

Parameters:

fp The standard io file handle to write to. ie. stderr

The documentation for this class was generated from the following files:

- iso8211.h
- ddfsubfielddefn.cpp