-0045

A Parser for the ISO 8211 Data Format

Michael McDonnell

January 1994

U.S. Army Corps of Engineers
Topographic Engineering Center
Fort Belvoir, Virginia 22060-5546

US Army Corps of Engineers
Topographic Engineering Center

TEC

xcnps

31635793

UG470
U57
no. TEC-0045

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE January 1994	3. REPORT TYPE AND DATES COVERED Technical Report

4. TITLE AND SUBTITLE A Parser for the ISO 8211 Data Format	5. FUNDING NUMBERS PR 4A161102B52C TA CO WU 014
6. AUTHOR(S) Michael McDonnell	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Topographic Engineering Center 7701 Telegraph Road Alexandria, VA 22310-3864	8. PERFORMING ORGANIZATION REPORT NUMBER TEC-0045
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of the Chief of Engineers Washington, DC 20314-1000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes a library of functions that parse an ISO 8211 file and convert the parsed data into a form useful to other programs, which can then read user data from the file. The structure of ISO 8211 files will be defined and then it will be shown how these programs interpret data in those files. Finally, an example program will be presented that reads an ISO 8211 file by using this library.

14. SUBJECT TERMS ISO 8211, data parsing, data conversion	15. NUMBER OF PAGES 24
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNLIMITED

CONTENTS

PREFACE

This study was conducted under DA Project 4A161102B52C, Artificial Intelligence Concepts for Terrain Analysis.

The study was conducted under the supervision of Mr. John Benton, Chief, Artificial Intelligence Division; and Mr. John Hansen, Director, Research Institute Laboratory, U.S. Army Topographic Engineering Center.

Walter E. Boge was Director, and Lt. Col. Louis R. DeSanzo was the Commander and Deputy Director of the U.S. Army Topographic Engineering Center at the time of publication of this report.

A PARSER FOR THE ISO 8211 DATA FORMAT

The ISO 8211 File Format

The International Standards Organization (ISO) has defined standard 8211 as a "data descriptive file for information interchange" [ISO85]. The ISO 8211 format is hierarchical. Terms referred to in the standard document ISO 8211-1985(E) will be used to refer to the sections of this hierarchy. Each file starts with a single Data Descriptive Record (DDR), which describes the formats of the Data Records (DR) that follow. There may be many DR sections. All user data is contained in the DR sections.

Both the DDR and each of the DR have a similar internal structure. Each is divided into three sections. Each begins with a 24-byte *leader* that gives the sizes of the sections that follow. The leaders are followed by a *directory* that gives the lengths and positions (offsets) of each of the data fields to be found in the final section. The final section is divided into *fields*, and each field is given a mnemonic alphanumeric *tag* for identification. These tags are defined in the directory section. The DDR and each DR have a similar directory section, but the final section of each of these differs. The final section of the DDR is called the *Data Descriptive Area* (DDA), and the final section of each DR is called the *User Data Area* (UDA). As its name implies, it is in the UDA that the actual data being transmitted by the ISO 8211 file is stored.

The appendix to this report contains a listing of the C programming language "include" file *iso8211.h* which defines data structures used in the parser. Refer to this listing for definitions of structures and constants mentioned in the following discussion.

The DDA

The DDA contains a succession of *fields*, which in turn have *subfields*. Fields are separated by a *field terminator*, which is the hexadecimal ASCII character 1e. Subfields are separated by a *unit terminator*, which is the hexadecimal ASCII character 1f. The subfields are, in order, *field controls*, *data field name*, *label*, and *format controls*. Not all of these subfields need exist. Missing subfields are indicated by a pair of consecutive field terminators. The mnemonic tags from the DDR directory are assigned to each field in turn, so the number of tags in the directory must be the same as the number of fields in the DDA. The tags are then referred to in each DR to connect the data in the DR with the data in the DDR.

Here is an example of a DDA field, which has been formatted for readability by breaking it into separate lines:

```
1600;&
TEST_PATCH_IDENTIFIER_FIELD&
PNM!DWV!REF!PUR!PIR!PIG!PIB&
(A(7),I(6),R(5),R(5),I(3),I(3),I(3));
```

The first line, "1600;&", is the *field controls*. The number tells what type of field this refers to (here a *mixed vector* field) and the characters ";" and "&" tell us that these characters may be used as printed representations of the field terminator and the unit terminator respectively, as is done in the listing above. The second line is an identifying *name* terminated by a unit terminator. The third line is a *label*, which is in this case a *vector label* consisting of a series of subfield labels for the DR. Each subfield label is separated from its neighbor by a "!". The fourth and final line has the *format controls* subfield which specifies the format of data in the UDA by using a FORTRAN-like syntax. The format controls subfield is delimited by a field terminator. The DDA field as a whole is also delimited by a field terminator.

For a vector label such as this example, each vector subfield tag is associated with one of the data formats in the format control. Therefore, subfield tag "PNM" refers to an ASCII string that is seven characters (bytes) long; tag "DWV" refers to an integer that is encoded as a string of six ASCII numeric characters; and tag "REF" refers to a floating point number encoded as a string of five ASCII numeric characters with floating point characters such as "." also allowed.

The UDA

The final section of each DR is called the User Data Area (UDA). It is here that the useful data being transmitted by the ISO 8211 file is stored. Following the leader, the directory section of each DR contains tags that must also appear in the DDR directory. These tags then index the corresponding entry in the DDA, which tells how to read the UDA by supplying formats. A given tag in the DDR may be referenced any number of times in a DR.

For all the details of ISO 8211, refer to the standards document. Enough of ISO 8211 has now been presented so that we can understand the important data structures used in this parser.

The Parser

This parser runs under the UNIX operating system, although it has been written to be portable to different environments. It works by building lists of structures that can then be interrogated by programs that need the data to read ISO 8211 files. The parser moves forward through the file being parsed. For example, when the parse of a DR directory is complete the file pointer is at the beginning of the associated UDA. Every parsing program takes a stream pointer to the file being read. In the C programming language this is declared as a (FILE *) type, such as

```
FILE *fp;
```

The parser turns input data into a set of lists. Lists are formed from C data structures linked together using a pointer that is part of the structure. This pointer is always named *next*. Lists are always terminated by a NULL pointer. Any type of list may then be traversed using C code such as the following, which traverses list "foo" by using a user-defined pointer named "foop":

```
for (foop = foo; foop != NULL; foop = foop->next)
{
        /* do something with elements of list foo */
}
```

All data structures used by the parser are defined in file *iso8211.h*, which is listed in the appendix. Data from the DDR is parsed into a list based on a C structure called *dda_entry* and data from each DR is parsed into a list based on a C structure called *uda_entry*. These two lists are the things that a user of the library will be concerned with.

To make the dda_entry list, call the function *parse_ddr()*, which returns a pointer to the head of the dda_entry list. Similarly, to parse the next DR, call function *parse_dr()*, which returns a pointer to the head of a uda_entry list. Each of these lists will now be discussed in turn.

The Parser: DDR Section

The dda_entry list is only parsed once since there is only one DDR in an ISO 8211 file. Function *parse_ddr()* takes a single argument, a pointer to the open ISO 8211 file being parsed, and returns a list of *dda_entry* structures. This list is then searched for matching tags when parsing each DR. The dda_entry structure is:

```
typedef struct dda_entry
{
 int    structure_type;  /* ELEMENTARY, VECTOR, ARRAY */
 int    data_type;              /* INT, FLOAT, EXP_FLOAT, ... */
 char   *name;                  /* long descriptive name */
 char   *tag;                   /* same as in corresponding ddr_entry */
 int    label_type;             /* VECT, CARTESIAN, ARRAY_DESC */
 union label *label;
 struct format *format;
 struct format *repeat;  /* indicate repeating part of format list */
 struct dda_entry *next;
}      dda_entry;
```

The first two members of this structure, *structure_type* and *data_type*, hold enumerated types, which can be found in the *iso8211.h* file. The third and fourth members, *name* and *tag*, refer to the long name for the entry and the short tag by which it will be referenced. The fifth member, *label_type* is an enumerated type code for the type of label found in the next member, *label*. The *label* member is a pointer to a union, which stores a type of data indexed by the *label_type* member:

```
typedef union label
{                                  /* a label will be one of three types */
 struct vector *vector;
 struct cartesian *cartesian;
 struct array_desc *desc;
}      label;
```

the *label* union can contain pointers to structures representing the three types of label supported under ISO 8211:

```
typedef struct vector
{
 char   *tag;
 struct vector *next;
}      vector;

typedef struct cartesian
{
 struct vector *rows;
 struct vector *cols;
 struct vectors *vecs;          /* higher dimensions if needed */
}      cartesian;

typedef struct array_desc
{
 int    length;          /* length of a dimension */
 struct array_desc *next;
}      array_desc;
```

The *cartesian* structure refers to a list of these structures:

```
typedef struct vectors
{                                  /* needed for cartesian labels more than
                                    * 2D */
 vector *vec;
 struct vectors *next;
}     vectors;
```

The *vectors* structure allows multidimensional arrays to be stored as lists of *vector* structures. Structure array_desc stores an *array descriptor*; a rather strange label that indicates the dimensions of an array, which will follow in the UDA. See the standard for an explanation of array descriptors.

The final two members of a *dda_entry* structure are pointers to a list of *format* structures:

```
typedef struct format
{
 int   type;                      /* INT, FLOAT, EXP_FLOAT, ... */
 int   length;           /* either this or delimiter must be \000 */
 char  delimiter;
 struct format *next;
}     format;
```

The UDA data may be delimited by either specifying its length or by specifying a delimiter character, which may not appear in the data itself. The *format* structure allows for each of these delimiting techniques, although at least one of the members *length* or *delimiter* must be zero (for this parser, binary zero is therefore not allowed as a delimiter). The 8211 standard says that if both length and delimiter are zero, the data elements are separated by unit terminators.

There are two format pointers in the *dda_entry* structure because the format is defined to implicitly repeat the last parenthesized expression at its right end. A repeat pointer is needed to allow data to be read using this implicit repeating format.

The Parser: DR Sections

As mentioned, there is only one DDR in an ISO 8211 file; so the DDR section only needs to be parsed once. There may be many DR sections however, so parsing of the DR is done by a separate program that is called as many times as needed. Program *parse_dr()* has one argument; a pointer to the file being parsed. *Parse_dr()* returns a list of uda_entry structures:

```
typedef struct uda_entry
{
 char  *field_tag;             /* length is up to field terminator */
 char  *vec_tag;               /* length is up to next vector item */
 char  type;                   /* A,I,R,S,C,B, or X */
 union {
  char *cp;                    /* CHAR (actually a string) */
  int i;                       /* INT */
  double d;                    /* FLOAT, EXP_FLOAT */
  int *bf;                     /* BITFIELD, CHAR_BIT_STRING */
  void *ignore;         /* IGNORE */
```

```
    }data;                      /* user data. */
    struct uda_entry *next;
  }     uda_entry;
```

The *field_tag* member corresponds to field tags in the dda_entry structure and is used to find a corresponding entry in the dda_entry list. Member *vec_tag* is one of the vector subfield tags mentioned above in the discussion of the DDA and is used to find the exact match for a format from the format list associated with each item in the dda_entry list. The *type* member is a character indicating the data type, which will be stored in this instance of the uda_entry structure. The *data* member is a union whose type is indexed by the *type* member.

Besides the high_level functions *parse_ddr()* and *parse_dr()*, there are lower level parsers available for those cases when more control is needed. These are

```
extern struct ddr_leader *parse_ddr_leader();
extern struct ddr_entry *parse_ddr_directory();
extern struct dda_entry *parse_dda();
```

which separately parse the three main sections of the DDR, and

```
extern struct dr_leader *parse_dr_leader();
extern struct dr_entry *parse_dr_directory();
```

which parse the first two sections of a DR. See file *iso8211.h* for definitions of the structures referred to in these function declarations. The UDA is too variable to support a parser in this library; the user of the library must define one. The code that follows gives an example of this.

An Example

Here is an example of C code that uses the programs *parse_ddr()* and *parse_dr()*:

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

main(argc, argv)
 int   argc;
 char **argv;
{
 struct dda_entry *dda = NULL;
 struct dr_entry *dr = NULL, *drp;
 FILE *fp;


 dda = parse_ddr(fp);
 while (1)                     /* do until EOF */
 {
  dr = parse_dr(fp);

  /* Do something in here with data from DR, if desired. */

  /* last dr element has the seek information we need to go past uda */
  for (drp = dr; drp->next != NULL; drp = drp->next)
   ;
/* for now, seek past the uda */
if (fseek(fp, (long)(drp->position + drp->length), 1) == -1)
  exit(0);

/*
 * No parse_uda() function is defined here because the
 * user data area (uda) may contain many types of structures and the
 * parse is therefore data-dependent.
 */
}
}
```

A Longer Example

As the last comment in the code above shows, there is no *parse_uda()* function defined in the library. The library takes care of those parts of ISO 8211 that are not data-dependent. The user of this library should write UDA parsers, as needed, based upon the information retrieved by the parsers described here. A final, rather long, example showing such usage is this section of code from a parser of ARC Digitized Raster Graphics (ADRG), a product of the U.S. Defense Mapping Agency:

```
...

/************* TRANSMITTAL HEADER FILE ************/
/*
 * Transmittal Header File always has the same name, so just open
 * the one in the current directory.  From the THF we want filenames
 * and the corners of the Distribution Rectangle in lat,lon.
 */
if ((fp = fopen("TRANSH01.THF", "r")) == NULL)
{
  fprintf(stderr, "File open error: %s\n", argv[1]);
  exit(1);
}
dda = parse_ddr(fp);              /* parse the file directory */

/* THF has 4 records.  First record contains corner coords of image. */
uda = parse_next_dr(dda, fp);
get_corners(&nw, &se, uda);

/* Next two dr records contain nothing of interest
 * (security and test patch respectively).
 */
parse_next_dr(dda, fp);
parse_next_dr(dda, fp);

/* Next (and last) dr record contains the filenames.  Build and
 * return a directory tree.
```

```
     */
    uda = parse_next_dr(dda, fp);
    root = parse_directory(uda);
    fclose(fp);

    /* Do similar things with other files */
    ...
```

The library function *parse_ddr()* is used by this code, but the programmer has encapsulated the *parse_dr()* function in a function of his own called *parse_next_dr()*:

```
/*
 * return a uda list associated with current dr.
 */

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

uda_entry *
parse_next_dr(dda, fp)
  dda_entry *dda;
  FILE   *fp;
{
  dr_entry *dr, *drp;
  dda_entry *dap;
  uda_entry *uda, *temp, *head = NULL;
  int c;
  extern uda_entry *parse_vec();
  extern uda_entry *parse_cart();
  extern uda_entry *parse_desc();

  dr = parse_dr(fp);                    /* parse leader and dr directory */
  for (drp = dr; drp != NULL; drp = drp->next)
  {
    for (dap = dda; dap != NULL && strcmp(drp->tag, dap->tag) != 0;
            dap = dap->next)
      ;                                 /* find match in dda to get format */
    if (dap == NULL)                    /* no match; an error */
    {
      fprintf(stderr, "No match found for tag %s\n", drp->tag);
      break;
    }
    switch (dap->label_type)
    {
    case 0:                             /* no label; there should be a function... */
      break;
    case VECT:
      uda = parse_vec(drp, dap, fp);
      break;
      case CARTESIAN:
        uda = parse_cart(drp, dap, fp);
        break;
      case ARRAY_DESC:
        uda = parse_desc(drp, dap, fp);
        break;
      default:
        fprintf(stderr, "No such label type:%d\n", dap->label_type);
        break;
      }
      if (head == NULL)
      {
        head = uda;
        for(temp = uda; temp->next != NULL; temp = temp->next)
              ;
      }
      else
        for(temp->next = uda; temp->next != NULL; temp = temp->next)
              ;
      while ((c = getc(fp)) == UNIT_TERM || c == FIELD_TERM)
        ;                                  /* test next char to see if should skip */
      ungetc(c, fp);
    }
    return head;
  }
```

Function *parse_next_dr()* in turn calls application-specific UDA parsers that were written to conform to the ADRG format. Here is one of them:

```
uda_entry *
parse_vec(dr, dda, fp)
  dr_entry *dr;
  dda_entry *dda;
  FILE   *fp;
{
  uda_entry *uda, *temp, *head = NULL;
  vector *vec;
  format *fmt;
  extern void get_data_value();

  for (vec = dda->label->vector, fmt = dda->format;
      fmt != NULL; fmt = fmt->next, vec = vec->next)
  {
    temp = (uda_entry *) malloc(sizeof(uda_entry));
    temp->vec_tag = malloc(strlen(vec->tag) + 1);
    strcpy(temp->vec_tag, vec->tag);
    temp->field_tag = malloc(strlen(dda->tag) + 1);
    strcpy(temp->field_tag, dda->tag);
```

```
   temp->next = NULL;
   get_data_value(fmt, temp, fp);          /* temp and fp changed */
   if (head == NULL)
     head = temp;
   else
     uda->next = temp;
   uda = temp;
  }
  return (head);
}
```

Function *parse_vec()* builds a list of structures specific to the ADRG format. It uses the lists of dda_entry and of uda_entry structures from the library functions *parse_ddr()* and *parse_dr()* to help in finding UDA data. Obtaining the UDA data is done by function *get_data_value()*:

```
void
get_data_value(fmt, uda, fp)
  format *fmt;
  uda_entry *uda;
  FILE  *fp;
{
  int    c;
  char   *buf;

  buf = malloc(fmt->length + 1);
  while ((c = getc(fp)) == UNIT_TERM || c == FIELD_TERM)
    ;                               /* test next char to see if should skip */
  ungetc(c, fp);
  if (fread(buf, 1, fmt->length, fp) != fmt->length)
  {
    fprintf(stderr, "Read error in get_data_value\n");
    return;
  }
  buf[fmt->length] = ' ';/* for string operations */
  uda->type = fmt->type;
  switch (fmt->type)
  {
  case 'I':                         /* integer */
    uda->data.i = atoi(buf);
    break;
  case 'A':                         /* (char *) */
    uda->data.cp = malloc(fmt->length + 1);
    strcpy(uda->data.cp, buf);
    break;
  case 'R':                         /* real number */
  case 'S':                         /* exponential real number */
    for (c = 0; buf[c] != ' '; ++c)
      if (buf[c] == 'D')     /* FORTRAN indicator of exponential */
```

```
            buf[c] = 'e';                    /* C indicator of exponential */
        uda->data.d = atof(buf);
        break;
      default:                               /* no other data type legal in adrg */
        fprintf(stderr, "Data type %d illegal\n", fmt->type);
        break;
      }
    }
```

This completes the example. Study of this example will show at least one way of using the basic data structures returned by *parse_ddr()* and *parse_dr()* to read the UDA. It is not advisable to use uda_entry structures for all user data. Large arrays, for example, should be read directly once other information has been extracted from the ISO 8211 file.

Discussion

The parser is used by including file *iso8211.h* in your program and linking the program with library *iso8211*. In a C program, this linking is done by a command such as:

```
cc -o myprog myprog.c -liso8211
```

The C programs to build the library are available from the author at the internet address of mike@tec.army.mil, or for anonymous ftp from pooh.tec.army.mil as compressed tar file pub/iso8211.tar.Z. There is no charge for this code. The programs are all unrestricted and in the public domain. The code also includes an example parser written using the library. Some of the programs used in this report are from the example parser.

Appendix

This is a listing of the include file *iso8211.h*. This file defines constants and data structures used in the parser and declares the functions available to a user in the *iso* library. The comments in the file explain each entry.

```
#ifndef ISO8211_H
#define ISO8211_H

/*
 * This file and associated programs were written by Mike McDonnell
 * of the U.S. Army Topographic Engineering Center (mike@tec.army.mil).
 * They are in the public domain. Please retain this comment.
 */

/* ddr and dr leaders are of a fixed length; 24 bytes. */
#define LEADER_LENGTH 24

/*
 * Field and unit terminators are used throughout ISO8211 files. The
 * term "unit" means a subfield within a larger field.
 */
#define FIELD_TERM ' 36'    /* ctrl-^ */
#define UNIT_TERM ' 37'     /* ctrl-_ */

/*
 * These are mnemonic macros showing what the various dda_entry.controls
 * data types are. Besides these numeric values, the trailing chars '&'
 * and/or ';' indicate that these printable chars may be used as
 * printed representations of UNIT_TERM and FIELD_TERM respectively.
 *
 * The ISO8211 document describes numeric data types as "implicit point"
 * for integers, "explicit point" for floats, and "scaled explicit
 * point" for floats in scientific notation. I have used the more
 * mnemonic names of "INT", "FLOAT", and "EXP_FLOAT" for these numeric
 * types.
 */

/* The first char is the structure type */
enum structure_type
{
 ELEMENTARY,
 VECTOR,
 ARRAY
};

/* The second char is the basic data type */
```

```
enum data_type
{
 CHAR,
 INT,
 FLOAT,
 EXP_FLOAT,
 CHAR_BIT_STRING,
 BITFIELD,
 IGNORE
};

/* label types; make numbers big to stay out of way of lex's defaults */
enum label_type
{
 VECT = 3,
 CARTESIAN = 4,
 ARRAY_DESC = 5
};

/*
 * The ISO 8211 file consists of a data descriptive record (ddr)
 * followed by data records (dr). This section describes the structures of
 * the ddr. The ddr in turn describes the structures of the dr.
 */

/*
 * The data definition record (ddr) leader is of fixed format; 24 bytes
 * long. I use a standard trick (for me) of defining an ascii struct to
 * overlay the data in the buffer as read and then define a
 * corresponding struct in which ascii elements are appropriately
 * converted.
 */
typedef struct ascii_ddr_leader
{
 char   record_length[5];          /* total length of ddr including
                                    * terminator */
 char   interchange_level[1];      /* 3 levels are defined; 1, 2, 3 */
 char   leader_id[1];              /* 'L' for ddr leader */
 char   extension_flag[1];         /* 'E' for extended char sets; else ' ' */
 char   res1[1];          /* reserved; ' ' for now */
 char   application_flag[1];       /* reserved; ' ' for now */
 char   field_control_length[2]; /* bytes in ddf for type and
                                          * struct codes; also used in
                                          * df? */
 char   dda_base[5];               /* offset of dda in ddr */
 char   extended[3];               /* specify extended char sets; else '
                                    * ' */
 char   length_size[1];  /* see below */
 char   position_size[1];         /* see below */
 char   res2[1];          /* reserved; '0' for now */
 char   tag_size[1];              /* see below */
}    ascii_ddr_leader;

typedef struct ddr_leader
{
 int    record_length;   /* total length of ddr including
                            * terminator */
 int    interchange_level;        /* 3 levels are defined; 1, 2, 3 */
 char   leader_id[2];             /* 'L' for ddr leader */
 char   extension_flag[2];        /* 'E' for extended char sets; else ' ' */
 char   res1[2];         /* reserved; ' ' for now */
 char   application_flag[2];      /* reserved; ' ' for now */
 int    field_control_length;     /* bytes in ddf for type and struct
                                   * codes */
 int    dda_base;                 /* offset of dda in ddr */
 char   extended[4];              /* specify extended char sets; else ' ' */
 int    length_size;              /* see below */
 int    position_size;   /* see below */
 int    res2;                     /* reserved; '0' for now */
 int    tag_size;        /* see below */
}    ddr_leader;

/*
 * Notice that many of these structs have a "next" pointer and so are
 * designed to make lists.  As a convention, I do not store the length
 * of these lists.  To find length of lists, just traverse them and
 * count the traversals.  This is not a very expensive operation and it
 * keeps the data structures simple.
 */

/*
 * A linked list of these structs constitutes the ddr directory. There
 * is a one-to-one correspondence between the ddr_entry structs and the
 * corresponding dda structs as described below. The ddr region is
 * terminated with a FIELD_TERM (ctrl-^).
 *
 * Field tags of 0 and 1 are reserved for the filename and the record ID
 * name respectively.  'length' is the total length of the dda field
 * (see below) including terminator characters. 'position' is the offset
 * of the dda field from the start of the dda area.
 */
typedef struct ddr_entry
{
 char   *tag;                     /* length gotten from tag_size in leader */
 int    length;          /* ascii length gotten from length_size
                            * in leader */
 int    position;           /* ascii length gotten from
                                    * position_size in leader */
 struct ddr_entry *next;
}     ddr_entry;


/*
 * This is the data descriptive area (dda) of the ddr.
 *
 * The length of the dda list will be the same as the length of the
 * ddr_directory list above.
 */

/*
 * Vector label tags are separated from each other by a '!' and formats
 * are in parentheses to be able to build up a tree structure as in
 * LISP. Format specification is as in FORTRAN with repeat specs like
 * 4I(7) to specify four integer fields of 7 ascii numeric characters
 * each.  See the standard for the (messy) details of the format spec.
 */

typedef struct vector
{
 char   *tag;
 struct vector *next;
}     vector;

typedef struct vectors
{                                  /* needed for cartesian labels more than
                                    * 2D */
 vector *vec;
 struct vectors *next;
}     vectors;

typedef struct cartesian
{
 struct vector *rows;
 struct vector *cols;
 struct vectors *vecs;             /* higher dimensions if needed */
}     cartesian;

typedef struct array_desc
{
 int    length;             /* length of a dimension */
 struct array_desc *next;
}     array_desc;

typedef union label
```

```
{                                  /* a label will be one of three types */
 struct vector *vector;
 struct cartesian *cartesian;
 struct array_desc *desc;
}     label;

/*
 * The format list will be circular at its end since it must
 * automatically repeat within the last set of parens. Rather than
 * actually make the list circular, I define a pointer to the repeating
 * part of the list, which always repeats to the end.
 *
 * An interesting twist in formats is found here in that data may be
 * delimited as well as of a fixed length. Thus A(,) means a string of
 * ASCII characters delimited by a comma. Data may be either delimited
 * or have a fixed length. Therefore at least one of the members
 * "length" or "delimiter" must be zero. They may also both be zero for
 * data delimited by UNIT_TERM.
 */
typedef struct format
{
 int    type;                      /* INT, FLOAT, EXP_FLOAT, ... */
 int    length;           /* either this or delimiter must be  00 */
 char   delimiter;
 struct format *next;
}     format;

typedef struct ascii_dda_entry
{
 char   *controls;                 /* length is gotten from header
                                    * field_control_length */
 char   *name;                     /* length up to terminator */
 char   *label;           /* length up to terminator */
 char   *format;          /* length up to terminator */
 struct ascii_dda_entry *next;
}     ascii_dda_entry;

typedef struct dda_entry
{
 int    structure_type;   /* ELEMENTARY, VECTOR, ARRAY */
 int    data_type;                 /* INT, FLOAT, EXP_FLOAT, ... */
 char   *name;                     /* long descriptive name */
 char   *tag;                      /* same as in corresponding ddr_entry */
 int    label_type;                /* VECT, CARTESIAN, ARRAY_DESC */
 union label *label;
 struct format *format;
 struct format *repeat;  /* indicate repeating part of format
                                    * list */
 struct dda_entry *next;
}      dda_entry;


/*
 * The ISO 8211 file consists of a data descriptive record (ddr)
 * followed by data records (dr).  This section describes the basic
 * structure of all dr.  The ddr describes the detailed structures of
 * each dr region. See above for data structures of the ddr.
 *
 * The data record (dr) leader is of fixed format; 24 bytes long.
 *
 * Standard trick here; make an all-ascii struct to overlay on the input
 * buffer and pick up the fields, then have another struct with the same
 * field names which are now integers, etc as appropriate. Note that
 * even single-character fields are saved as strings so that strncmp()
 * may be used consistently for all comparisons.
 */

typedef struct ascii_dr_leader
{
 char   record_length[5];       /* total length of dr */
 char   res1[1];        /* reserved; ' ' for now */
 char   leader_id[1];           /* 'D' for once; 'R' for repeat */
 char   res2[5];        /* reserved; 5 spaces '     ' for now */
 char   data_base[5];           /* offset of user data area (uda) in dr */
 char   res3[3];        /* reserved; 3 spaces '   ' for now */
 char   length_size[1]; /* see below */
 char   position_size[1];       /* see below */
 char   res4[1];        /* reserved; '0' for now */
 char   tag_size[1];            /* see below */
}      ascii_dr_leader;

typedef struct dr_leader
{
 int    record_length;  /* total length of dr */
 char   res1[2];        /* reserved; ' ' for now */
 char   leader_id[2];           /* 'D' for once; 'R' for repeat */
 char   res2[6];        /* reserved; 5 spaces '     ' for now */
 int    data_base;              /* offset of user data area (uda) in dr */
 char   res3[4];        /* reserved; 3 spaces '   ' for now */
 int    length_size;            /* see below */
 int    position_size;  /* see below */
 int    res4;                   /* reserved; '0' for now */
 int    tag_size;       /* see below */
}      dr_leader;
```

```
/*
 * A linked list of these structs constitutes the dr directory. There is
 * a correspondence between the dr_entry structs and the
 * uda (user data area) structs as described below.
 * Corresponding structs are matched by the "key" member in dr_entry
 * and the "field_tag" member in uda_entry.  The
 * directory region is terminated with a FIELD_TERM.
 *
 * 'length' is the total length of the uda field (see below) including
 * terminator characters. 'position' is the offset of the uda field from
 * the start of the uda area.
 *
 * This is exactly the same as a ddr_entry struct.  I may combine them
 * some day, I just didn't realize that they were the same until I was
 * done with the parser.  Keeping them separate makes it easier to
 * keep the names of things separate anyway.
 */
typedef struct dr_entry
{
 char  *tag;                  /* length gotten from tag_size in leader */
 int   length;         /* length of "length" is gotten from
                              * length_size in leader */
 int   position;       /* length is gotten from position_size
                              * in leader */
 struct dr_entry *next;
}     dr_entry;

/*
 * This is the user data area (uda) of the dr.
 * The length of the uda list will be the same as the length of the
 * dr_entry list above. Each entry in the uda is also terminated
 * with a FIELD_TERM (ctrl-^).
 *
 * The only thing that might have to be
 * handled specially in here is if arrays are defined by an array
 * descriptor in the uda; a strange beast that is just like an array
 * descriptor as may be found in the dda label field except that it
 * has its fields separated by UNIT_TERM (ctrl-_) instead of commas.
 */
typedef struct uda_entry
{
 char  *field_tag;            /* length is up to field terminator */
 char  *vec_tag;              /* length is up to next vector item */
 char   type;          /* A,I,R,S,C,B, or X */
 union {
  char *cp;                   /* CHAR (actually a string) */
  int i;               /* INT */
  double d;                   /* FLOAT, EXP_FLOAT */
```

```
   int *bf;                          /* BITFIELD, CHAR_BIT_STRING */
   void *ignore;             /* IGNORE */
  }data;                     /* user data. */
  struct uda_entry *next;
}      uda_entry;


extern char *malloc();             /* Have to put this somewhere. */
extern format *formatlist;         /* global pointer where format list goes */
extern format *repeatlist;         /* global pointer for format list repeat */

/*
 * All the public functions
 */
extern struct ddr_leader *parse_ddr_leader();
extern struct ddr_entry *parse_ddr_directory();
extern struct dda_entry *parse_dda();
extern struct dda_entry *parse_ddr();
extern struct dr_leader *parse_dr_leader();
extern struct dr_entry *parse_dr_directory();
extern struct dr_entry *parse_dr();

#endif                             /* ISO8211_H */
```

REFERENCE

Information Processing - Specification for a data descriptive file for information interchange, International Organization for Standardization publication ISO 8211-1985(E), 15 Dec 1985.