CLARION TECHNICAL BULLETIN

Bulletin #117

Clarion Data Files

Overview

This bulletin explains the format of Clarion data files.

Clarion Data Files

This technical bulletin explains the format of data files created by Clarion Professional and Personal Developer (version 2 and above). It will only cover data files; key and index files will be covered in a separate technical bulletin. Use this information carefully because an erroneous change to a data file will make your file unreadable. It will not cover encryption because if we told you how to un-encrypt a file, other people would be able to do the same, thus risking the integrity of your data.

Now, just what is in those files?

When you define a file with Clarion, you provide a lot of information about your data: information on fields, keys, indexes, pictures, arrays, etc. When your file is first created, you will have a .DAT file and, optionally, one or more .K??, .I??, and .MEM files (where ?? is a two digit number). The .DAT file contains more than your data; it also contains information on your keys, fields and their layouts, and many fields that allow Professional Developer to efficiently use the space on your disk.

Let's look at the .DAT file first. It consists of several sections, some of which are optional. They are:

1. the file header
2. field descriptors
3. key and index descriptors
4. picture descriptors
5. array descriptors
6. your data

The file header and field descriptors are always in the file. The other sections exist only if they are needed. Obviously, if you don't have any data in the file, then section 6 is not needed.

Let's look at each section individually. To avoid confusion, the picture and array descriptors will be deferred until later in this bulletin.

1. The file header

The file header consists of 23 fields, many of which tell Professional Developer what else it is going to have to look for in the rest of the .DAT file. Here is a C structure that defines the file header:

```
struct {
   unsigned filesig;          /* file signature */
   unsigned sfatr;            /* file attribute and status */
   unsigned char numbkeys;    /* number of keys in file */
   unsigned long numrecs;     /* number of records in file */
   unsigned long numdels;     /* number of deleted records */
   unsigned numflds;          /* number of fields */
   unsigned numpics;          /* number of pictures */
   unsigned numarrs;          /* number of array descriptors */
   unsigned reclen;           /* record length (including record header) */
   unsigned long offset;      /* start of data area */
   unsigned long logeof;      /* logical end of file */
   unsigned long logbof;      /* logical beginning of file */
   unsigned long freerec;     /* first usable deleted record */
   unsigned char recname[12]; /* record name without prefix */
```



```
    unsigned char memnam[12];  /* memo name without prefix */
    unsigned char filpre[3];   /* file name prefix */
    unsigned char recpre[3];   /* record name prefix */
    unsigned memolen;          /* size of memo */
    unsigned memowid;          /* column width of memo */
    unsigned long reserved1;   /* reserved */
    unsigned long chgtime;     /* time of last change */
    unsigned long chgdate;     /* date of last change */
    unsigned reserved2;        /* reserved */
};
```

When looking at a dump of a .DAT file, just remember this: the fields marked "unsigned" (otherwise known as "unsigned int") take up two bytes. The "unsigned char" fields take up one byte. The "unsigned long" fields take up 4 bytes. Also remember that the bytes of unsigned longs and unsigned ints will be reversed on disk, due to the way the Intel CPUs store their data.

Almost all of the fields in the header are simply numbers or strings. One field, however, is a bit-map. The file attribute field (sfatr) is a 2-byte field. Currently, only the lower 8 bits are used. The following bits are either turned on or off, depending on what is currently happening to the file. Bit 0 is the low order bit and bit 15 is the high order bit. If a bit is turned on, then the corresponding condition is true for the file:

bit 0 - file is locked
bit 1 - file is owned
bit 2 - records are encrypted
bit 3 - memo file exists
bit 4 - file is compressed
bit 5 - reclaim deleted records
bit 6 - file is read only
bit 7 - file may be created

2. The field descriptors

Following the file header are the field descriptors. There are as many field descriptor entries as there are fields in the file. Each field descriptor consists of 8 fields. They are defined as follows:

```
struct {
   unsigned char fldtype; /* type of field */
   char fldname[16];      /* name of field */
   unsigned foffset;      /* offset into record */
   unsigned length;       /* length of field */
   unsigned char decsig;  /* significance for decimals */
   unsigned char decdec;  /* number of decimal places */
   unsigned arrnum;       /* array number */
   unsigned picnum;       /* picture number */
};
```

The "fldtype" field tells Professional what type of field this is. The following types are currently used:

1 - LONG
2 - REAL
3 - STRING
4 - STRING WITH PICTURE TOKEN
5 - BYTE
6 - SHORT

7 - GROUP
8 - DECIMAL

3. The key and index descriptors

If you defined any keys for your file, then the field descriptors will be followed by the key descriptors. Like the field descriptors, there is one entry in the key descriptors for each key/index you defined. Key descriptors are broken up into two parts: **keysects** and **keyparts**.

They are defined as follows:

```
struct {
   unsigned char numcomps; /* number of components for key */
   char keynams[16];         /* name of this key */
   unsigned char comptyp;  /* type of composite */
   unsigned char complen;    /* length of composite */
} KEYSECT;

struct {
   unsigned char fldtype;    /* type of field */
   unsigned fldnum;        /* field number */
   unsigned elmoff;          /* record offset of this element */
   unsigned char elmlen;     /* length of element */
} KEYPART;
```

There is exactly one KEYSECT for each key/index you define. There is a KEYPART for each field component in a key or index.

4. The picture descriptors and 5. The array descriptors

We'll defer discussion of these two sections until later in this technical bulletin.

6. Your data

Finally, you get to your data. Each record of your data is preceded by a header. This header is defined as follows:

```
struct {
   unsigned char rhd;      /* record header type and status */
   unsigned long rptr;     /* pointer for next deleted record or memo if active */
};
```

"rhd" is a bit field that tells Professional developer the status of this record:

bit 0 - new record
bit 1 - old record
bit 2 - revised record
bit 4 - deleted record
bit 6 - record held

This header is followed by the fields you have defined for your records.

The last section contained quite a bit of information, but not all of the fields were completely explained. Let's clear up some confusion and look at an actual Clarion data file.

For this example, we'll use a simple file. Taken from the EXAMPLES subdirectory of Professional Developer, it is a small file called PHONEBK. It was defined in Professional Developer as follows:

```
FILE, PRE(PHN), RECLAIM, CREATE
   PHN:BY_NAME      KEY(PHN:NAME),DUP,NOCASE
   PHN:BY_COMPANY   KEY(PHN:COMPANY),DUP,NOCASE
   RECORD           RECORD
   NAME                STRING(30)
   COMPANY             STRING(30)
   ADDRESS             STRING(30)
   CITY                STRING(28)
   STATE               STRING(2)
   ZIP                 STRING(6)
   PHONE               DECIMAL(11)
                    . .
```

As you can see, this file has 7 fields, 6 of which are simple strings. Two keys are defined. Deleted records will be reclaimed by the system. Both keys are case-insensitive and allow duplicates. This file has no memo fields, is not encrypted, and has no composite keys. Let's take a look at each section of this file. First, since this file only has two records, the next page contains a dump of the entire file:

```
00000: 43 33 A0 00 02 02 00 00 00 00 00 00 00 07 00 00|C3..............|
00010: 00 00 00 89 00 44 01 00 00 02 00 00 00 01 00 00|.....D..........|
00020: 00 00 00 00 00 52 45 43 4F 52 44 20 20 20 20 20|.....RECORD     |
00030: 20 20 20 20 20 20 20 20 20 20 20 20 20 50 48 4E|             PHN|
00040: 20 20 20 00 00 00 00 00 00 00 00 9B E4 4F 00 1C|   ..........O..|
00050: 0D 01 00 00 00 03 50 48 4E 3A 4E 41 4D 45 20 20|......PHN:NAME  |
00060: 20 20 20 20 20 20 00 00 1E 00 00 00 00 00 00 00|      ..........|
00070: 03 50 48 4E 3A 43 4F 4D 50 41 4E 59 20 20 20 20|.PHN:COMPANY    |
00080: 20 1E 00 1E 00 00 00 00 00 00 00 03 50 48 4E 3A| ...........PHN:|
00090: 41 44 44 52 45 53 53 20 20 20 20 20 3C 00 1E 00|ADDRESS     <...|
000A0: 00 00 00 00 00 00 03 50 48 4E 3A 43 49 54 59 20|.......PHN:CITY |
000B0: 20 20 20 20 20 20 20 5A 00 1C 00 00 00 00 00 00|       Z........|
000C0: 00 03 50 48 4E 3A 53 54 41 54 45 20 20 20 20 20|..PHN:STATE     |
000D0: 20 20 76 00 02 00 00 00 00 00 00 00 03 50 48 4E|  v..........PHN|
000E0: 3A 5A 49 50 20 20 20 20 20 20 20 20 20 78 00 06|:ZIP          x..|
000F0: 00 00 00 00 00 00 00 08 50 48 4E 3A 50 48 4F 4E|........PHN:PHON|
00100: 45 20 20 20 20 20 20 20 7E 00 06 00 0B 00 00 00|E       ~........|
00110: 00 00 01 50 48 4E 3A 42 59 5F 4E 41 4D 45 20 20|...PHN:BY_NAME  |
00120: 20 20 20 70 1E 03 01 00 00 00 1E 01 50 48 4E 3A|   p........PHN:|
00130: 42 59 5F 43 4F 4D 50 41 4E 59 20 20 70 1E 03 02|BY_COMPANY  p...|
00140: 00 1E 00 1E 01 00 00 00 00 4D 61 72 6B 20 45 2E|.........Mark E.|
00150: 20 44 61 76 69 64 73 6F 6E 20 20 20 20 20 20 20| Davidson       |
00160: 20 20 20 20 20 20 20 43 6C 61 72 69 6F 6E 20 53|       Clarion S|
00170: 6F 66 74 77 61 72 65 20 20 20 20 20 20 20 20 20|oftware         |
00180: 20 20 20 20 20 31 35 30 20 45 2E 20 53 61 6D 70|     150 E. Samp|
00190: 6C 65 20 52 6F 61 64 2C 20 53 75 69 74 65 20 32|le Road, Suite 2|
001A0: 30 30 20 50 6F 6D 70 61 6E 6F 20 42 65 61 63 68|00 Pompano Beach|
001B0: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 46|               F|
001C0: 4C 33 33 30 36 34 20 00 30 57 85 45 55 01 00 00|L33064 .0W.EU...|
001D0: 00 00 52 61 79 20 50 69 64 67 65 20 20 20 20 20|..Ray Pidge     |
001E0: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20|                |
001F0: 50 72 6F 78 69 6D 69 74 79 20 54 65 63 68 6E 6F|Proximity Techno|
00200: 6C 6F 67 79 20 20 20 20 20 20 20 20 20 20 33 35|logy          35|
00210: 31 31 20 4E 45 20 32 32 6E 64 20 41 76 65 6E 75|11 NE 22nd Avenu|
00220: 65 20 20 20 20 20 20 20 20 20 20 20 20 46 6F 72 74|e           Fort|
00230: 20 4C 61 75 64 65 72 64 61 6C 65 20 20 20 20 20| Lauderdale     |
00240: 20 20 20 20 20 20 20 20 46 4C 33 33 30 36 33 20|        FL33063 |
00250: 00 30 55 66 35 11                              |.0Uf5. |
```

The file header is the first part of the file. Here is a breakdown of the file header, field by field (preceding each field name is the offset in hexadecimal for this file, assuming that the file starts at offset 0000; if you are using Scanner to look at the file, add one to each offset as Scanner starts the file at offset 0001).

(0000) filesig — the file's signature (43 33). This field tells us that this is a file created by Professional Developer version 2. Remember, since the bytes are stored in a reverse order due to the Intel architecture, this value is actually hex 3343.

(0002) sfatr — the file's attributes (A0 00). This is a bit-field, so let's convert it to binary. Hex 00A0 = 10100000 (the top byte is not used). Bits 7 and 5 are turned on. Referring back to the discussion of the sfatr field, you can see that this means "file may be created" and "reclaim deleted records."

(0004) numbkeys — the number of keys (02). This tells us this file has two key fields.

(0005) numrecs — the number of records (02 00 00 00). This file only has two records.

(0009) numdels — the number of deleted records (00 00 00 00). This file has no deleted records.

(000D) numflds — the number of fields defined in this file (07 00). This file has a total of 7 fields.

(000F) numpics — the number of pictures for this file (00 00). This file has no pictures defined and thus will not have a picture description section.

(0011) numarrs — the number of arrays for this file (00 00). This file has no arrays defined and thus will not have an array description section.

(0013) reclen — the record length, including the header (89 00). Hex 0089 equals 137 decimal. Our fields take up 132 bytes (30 + 30 + 30 + 28 + 2 + 6 + 6). Clarion's record header adds 5 bytes to this. Note: the DECIMAL(11,0) field takes up 6 bytes.

(0015) offset — the start of the data area (44 01 00 00). Our data begins 0144 hex bytes into the file.

(0019) logeof — the logical end-of-file (02 00 00 00). Our file logically ends at record 2.

(001D) logbof — the logical beginning-of-file (01 00 00 00). Our file logically begins at record 1.

(0021) freerec — the first usable deleted record (00 00 00 00). Our file has no deleted records, so this field is zero.

(0025) recnam — the record name without the prefix (52 45 43 4F 52 44 20 20 20 20 20 20 = "RECORD "). Notice that strings are padded with spaces and have no terminating NULL to indicate end-of-string.

(0031) memnam — the memo name without the prefix (20 20 20 20 20 20 20 20 20 20 20 20). There is no memo defined for this file, so this field is all blanks.

(003D) filpre — the file name prefix (50 48 4E = "PHN").

(0040) recpre — the record prefix (20 20 20). There is no record prefix, so this field is all blanks.

(0043) memolen — the size of the memo file (00 00). This file has no memo field, so this field has a value of 0.

(0045) memowid — the column width of the memo file (00 00).

(0047) reserved1 — (00 00 00 00). This field is reserved.

(004B) chgtime — the time of the last change to this file (9B E4 4F 00). 02:32 PM; this is stored in an "absolute" format, which is explained below.

(004F) chgdate — the date of the last change to this file (1C 0D 01 00). 08/11/89; this is also stored in an "absolute" format. See below.

(0053) reserved2 — (00 00). This field is also reserved.

That takes care of the file header. Next comes the field descriptors. There is one entry for each field in the file.

Here is a breakdown of the first field descriptor:

(0055) fldtype — the type of this field (03). Referring back to the discussion of the possible fldtype values, we can see that this is a STRING.

(0056) fldname — the name of this field (50 48 4E 3A 4E 41 4D 45 20 20 20 20 20 20 20 20 = "PHN:NAME"). Notice that the field name includes the prefix and is padded with spaces.

(0066) foffset — the offset into each record for this field (00 00). Since this is the first field in each record, it has an offset of 0, i.e. it is at the beginning of each record.

(0068) length — the length of the field (1E 00). This is the length of the field (in bytes). Hex 1E = 30 decimal, so this field is 30 bytes long.

(006A) decsig — the number of decimal places in this field (00). Since this is not a numeric field, this field is 0.

(006B) decdec — the significance for decimals (00). Since this is not a numeric field, this field is 0.

(006C) arrnum — the array number for this field (00 00). This field is not defined as an array, so this field is 0.

(006E) picnum — the picture number for this field (00 00). No picture number is associated with this field, so this field is 0.

The entries for fields 2 through 7 follow this one. They are all similar to this one, except that the fldname, foffset and length fields may have different values. Since field 7 is a DECIMAL, it's fldtype will be 08 and its length will be its actual length in the file, which is 6 bytes.

Following the field descriptors are the key descriptors. Like the field descriptors, there is one key descriptor for each key defined for your file. In this case, there are two key descriptors. Here is the breakdown of the first one (remember that the key descriptors are broken into two sections, KEYSECTS and KEYPARTS):

(0112) numcomps — the number of components for this key (01). This key is not a component key (it is composed of only one field), so this field has a value of 1.

(0113) keynams — the name of this key (50 48 4E 3A 42 59 5F 4E 41 4D 45 20 20 20 20 20 = "PHN:BY_NAME"). This is the name of the key, as defined to Professional. Note that it is padded with spaces.

(0123) comptyp — the type of the composite key (70). This is a bitfield that tells Professional certain things about this key. Since this field is only useful if you're going to be processing keys, it will be skipped. See the technical bulletin on key/index files for more info. Suffice to say that this field tells Professional that this is a key file (not an index file), duplicate entries are allowed, and case does not matter in keys.

(0124) complen — the length of the composite key (1E). Hex 1E = 30 decimal, which is the length of this key.

(0125) fldtype — the type of field this key is based on (03). This field uses the same values as defined above for fldtype in the field descriptor. Referring to that section, you can see that 03 means that this is a STRING.

(0126) fldnum — the field number this key is based on (01 00). This key is based on field #1 in the file.

(0128) elmoff — the offset into the record for the field this key is based on (00 00). Since this key is based on the first field in each record, the offset is 0.

(012A) elmlen — the length of the element in the file this key is based on (1E).

Following this key descriptor is the descriptor for key number 2. It is similar, except for the keynams, fldnum, and elmoff fields.

Finally, after all this, comes your data. However, Clarion's file system has added 5 bytes to the front of your records. For this first record, the header looks like this:

(0144) rhd — the record header type and status (01). Recall from the previous discussion of the header that 01 (binary 00000001) means "new record." Generally, you should not be concerned with this field unless it is set to "deleted record" or "locked record."

(0145) rptr — the pointer to the memo (00 00 00 00). This file has no memos, so this field is set to 0. If this record were deleted, this field would point to the next deleted record.

The record header for record 2 is exactly the same as for record 1. Each field is stored with no separators between them. STRINGs are padded with spaces out to their maximum defined length. DECIMALs are stored in a packed BCD-like format, with two digits per byte. Remember our DECIMAL field was defined as DECIMAL(11), so it will take up 6 bytes (2 digits per byte, with a half byte of padding since 11 won't divide by 2 evenly). Thus, our DECIMAL field looks like this for the first record:
00 30 57 85 45 55 = 305-785-4555. Even though it would fit in 5 bytes, we have to use 6 because we told Clarion it would take 11 digits. However, Clarion removes the separators in the number. We simply use a picture so it will display correctly.

That wasn't too difficult. But wait, you're asking, what about the key files? Unfortunately, even simple key files are rather hard to explain. Clarion products use a modified version of a B+ tree (named after the man who created them, R. Bayer). Instead of trying to explain exactly how key files work, I'll just cover the basics.

Key and Index files

Key files and Index files are read and written in blocks of 512 bytes. A key file has a 512 byte header that contains information about the structure of the key file itself. Following this header are a series of 512 byte nodes, which allow your programs to process a data file in key order. Even though the header of the key file occupies 512 bytes, currently only 35 bytes are used. The header of a key is laid out like this:

```
struct {
   unsigned long root;     /* number of root node */
   unsigned long numkent;  /* number of key entries */
   unsigned long numnode;  /* number of nodes for this index */
   unsigned long lastnod;  /* node number of last node */
   unsigned long keyeof;   /* record # of end of file */
   unsigned long keybof;   /* record # of beginning of file */
   unsigned long unused;   /* first unused node of file */
   unsigned char keytyp;   /* type of key */
   unsigned char keynode;  /* # of keys per node */
   unsigned char numcmps;  /* # of components of key */
   unsigned keylen;        /* total length of key entry */
```



```
    unsigned numlvls;       /* number of levels */
    char cvoid[477];        /* reserved space */
};
```

As mentioned above, it is not a good idea to go messing around with key files. If you absolutely need to know how key files are laid out, you'll probably have to delve into the technical bulletin that concerns how key/index files work.

Another aspect of key files that makes them difficult is the way Clarion stores the actual keys themselves. The keys are kept in a truly "sortable" order. This involves a lot of byte and bit flipping.

If you want to read a data base in a certain order, you could use Sorter to sort the file before reading it.

Dates and Times in the header

As mentioned previously, the dates and times in the header are stored in an "absolute" format. Here's how (in Clarion) to convert an absolute date and time to a more readable number.

First, the time:

Given an absolute time (abstim):

```
IF abstime < 1 OR abstime > 8640000
THEN
   STOP   ! abstime is invalid
ELSE
   abstime = abstime - 1
   hour = abstime / 360000
   abstime = abstime % 360000
   minute = abstime / 6000
   abstime = abstime % 6000
   seconds = abstime / 100
   hundreds = abstime % 100
```

The above algorithm assumes integer arithmetic; abstime is stored in a "long integer" field, which occupies 4 bytes. Here is an example:

Let's take the time from the file header mentioned above. The "time of last change" for the file is hex 9B E4 4F 00. First, reverse the order of the bytes (remember that bytes are stored low-byte, then high byte; in a long integer, the bytes of the integer are also flipped), which gives us hex 00 4F E4 9B.

1. Start with the hex value 00 4F E4 9B.
2. Convert this to decimal, giving us 5235867.
3. Compute 5235867 / 360000 = 14 with a remainder of 195867. 14 is the hour.
4. Compute 195867 / 6000 = 32 with a remainder of 3867. 32 is the minute.
5. Compute 3867 / 100 = 38 with a remainder of 67. 38 is the seconds; 67 is the hundredths of seconds.

Thus the time is 14:32:38.67, which is why the description above resulted in 2:32 PM as the last time of change for the file.

Dates are a little more involved. Given an absolute date (absday):

```
IF absday <= 3 OR absday > 109211
THEN
   STOP   ! absday is invalid
ELSE
   IF absday > 36527
   THEN
      absday = absday - 3
   ELSE
      absday = absday - 4
      .
   year = (1801 + (4 * (absday / 1461)))
   absday = absday % 1461
   IF absday != 1460
   THEN
      year = year + (absday / 365)
      day = absday % 365
   ELSE
      year = year + 3
      day = 365
      .
   IF year < 100
   THEN
      year = year + 1900
      .
   IF year % 4 = 0 AND year != 1900
   THEN
      number_of_days_in_February = 29
   ELSE
      number_of_days_in_February = 28
      .
   LOOP i = 1 TO 12 BY 1
      day = day - number_of_days_in_month_i
      IF day < 0
      THEN
         day = day + number_of_days_in_month_i + 1
         BREAK
      .
   month = i
end if
```

That's certainly more complicated than the time. The 'number_of_days_in_month_i' is a nonsense variable equal to the number of days in a particular month i, with i running from 1 (January) to 12 (December). Let's go through an example, again from the file header shown above. The date for this file is, again, a "long integer," stored on disk as hexadecimal 1C 0D 01 00. Reversing the order of the bytes gives us 00 01 0D 1C hex.

1. Start with the hex value 00 01 0D 1C.
2. Convert this to decimal, giving us 68892.
3. 68892 is > 36527, so subtract 3, giving us 68889.
4. Compute (1801 + (4 * (68889 / 1461))), giving 1989. 1989 is the year.
5. Compute 68889 % 1461, giving 222.
6. 222 is != 1460, so compute 222 / 365, giving 0. Add 0 to 1989 (the year).
7. Compute 222 % 365, giving 222. This is the day.
8. 1989 % 4 is != 0, so this date is not in a leap year. This means that February has 28 days.
9. Start our loop from 1 to 12, subtracting the number of days in each month from 'day' until 'day' drops below 0 (the value of i in the far right column is the current value of our loop variable).

January: day = day - 31, giving 191. (i = 1)
February: day = day - 28, giving 163.(i = 2)
March: day = day - 31, giving 132. (i = 3)
April: day = day - 30, giving 102. (i = 4)
May: day = day - 31, giving 71. (i = 5)
June: day = day - 30, giving 41. (i = 6)
July: day = day - 31, giving 10. (i = 7)
August: day = day - 31, giving -21. (i = 8)

10. Compute day = day + 31 + 1, giving 11. This is the actual day of the month. Since the loop terminated at i = 8, we know 8 is the month number. Thus, our date is 08/11/89.

Several items were left out of the prior discussion, in order to keep confusion to a minimum. Picture descriptors, array descriptors, composite keys, and memo fields were not covered at all. Let's take a look at each of these and see how they affect our file layout.

Composite Keys

Recall from the discussion above that key descriptors are made up of two parts: KEYSECTs and KEYPARTs. The example above had one of each. When a composite key is used, there is still only one KEYSECT in the key descriptors, but there will be one KEYPART for each field that makes up the component key. For example, if you have a key that is composed of the PHN:NAME and PHN:COMPANY fields, your key descriptor would have three pieces, not two. You would still have a KEYSECT, but the "numcomps" field would have a value of 2. The "complen" field would be equal to the sum of the lengths of the PHN:NAME and PHN:COMPANY fields.

Next, you would have two KEYPARTs, one for the NAME field and one for the COMPANY field. The only difficulty with handling composite keys is that you have to read more than one field from each record in order to build your key.

Picture Descriptors

If you have assigned a picture to a field in your database, then a picture descriptor will be created for that field. Picture descriptors follow the key descriptors in your file. They are laid out like this:

```
struct {
   unsigned piclen;
   char     picstr[256];
};
```

"piclen" contains the actual length of the picture since "picstr" (like all of the other strings in data files) is not null-terminated. If a picture descriptor has been created, the number associated with that particular picture descriptor will be placed in the "picnum" field in the field descriptor associated with that picture. Picture descriptor numbering starts at 1. Thus, if you are reading the field descriptors and the "picnum" field is not zero, you need to remember to look for the picture descriptor. This is due to the picture descriptor taking space in the file before the actual data records occur.

Array descriptors

Array descriptors look like this:

```
struct {
   unsigned numdim;        /* dims for current field */
   unsigned totdim;        /* total number of dims for field */
   unsigned elmsiz;        /* total size of current field */
   struct {
      unsigned maxdim;     /* number of dims for array part */
      unsigned lendim;     /* length of field */
   } ARRPART[sizeof(ARRPART)*totdim];
} ARRDESC;
```

Although this looks complicated, it really isn't. Each array descriptor consists of the "numdim," "totdim," and the "elmsiz" fields, followed by one or more "ARRPART" structures. There is one "ARRPART" for each dimension in the array.

Let's take the simplest case first. A single array causes one array descriptor to be created. Assuming the allocation STRING(10),DIM(3), the following array descriptor would be created:

```
struct {
   unsigned numdim = 1;
   unsigned totdim = 1;
   unsigned elmsiz = 30;
   struct {
      unsigned maxdim = 3;
      unsigned lendim = 10;
   };
};
```

"numdim" and "totdim" tell you whether or not this descriptor is part of another array. If they are equal, then this array descriptor stands by itself. "elmsiz" is the total size of the elements of this array. "maxdim" tells you the highest value allowed as the dimension. "lendim" is the length of each dimension. Thus, this array has 3 "elements", each of which is 10 bytes, giving an element size of 30. There is only one ARRPART structure since this array only has one dimension (3). To make this more complicated, let's add a group specifier.

```
GROUP,DIM(5)
   STRING(10),DIM(3)
```

This will cause two array descriptors to be created; one for the group and one for the array itself. One is created for the group because the group itself has a dimension. The first one (for the group) looks like this:

```
struct {
   unsigned numdim = 1;
   unsigned totdim = 1;
   unsigned elmsiz = 150;     /* 5 * 30 */
   struct {
      unsigned maxdim = 5;
      unsigned lendim = 30;
   };
};
```

The group has one dimension (5), with each element being 30 bytes, giving a total length of 150 bytes. Following this is a second array descriptor, this one being for the string itself.

```
struct {
   unsigned numdim = 1;
   unsigned totdim = 2;
   unsigned elmsiz = 30;
   struct {
      unsigned maxdim = 5;
      unsigned lendim = 30;
   };
   struct {
      unsigned maxdim = 3;
      unsigned lendim = 10;
   };
};
```

Notice that this array descriptor has two ARRPARTs. One covers the array of 5 that is the group itself; the second one covers the array of strings. Here's one more example. Let's get more complicated.

```
GROUP,DIM(5)
   STRING(10),DIM(3,6)
```

We will still have only two array descriptors. Again, the first one is for the group itself:

```
struct {
   unsigned numdim = 1;
   unsigned totdim = 1;
   unsigned elmsiz = 900;     /* 3 * 6 * 10 * 5 */
   struct {
      unsigned maxdim = 5;
      unsigned lendim = 180; /* 3 * 6 * 10 */
   };
};
```

Following this is the array descriptor for the strings themselves:

```
struct {
   unsigned numdim = 2;
   unsigned totdim = 3;
   unsigned elmsiz = 180; /* 3 * 6 * 10 */
   struct {
      unsigned maxdim = 5;
      unsigned lendim = 180;
   };
   struct {
      unsigned maxdim = 3;
      unsigned lendim = 60;
   };
   struct {
      unsigned maxdim = 6;
      unsigned lendim = 10;
   };
};
```

Array descriptors, like picture descriptors, are numbered. The array descriptors occur following the picture descriptors. If an array descriptor has been created for a field, the number assigned to the array descriptor will be in the "arrnum" field of the corresponding field descriptor.

Memo fields

Memo fields are unique in that they are stored in a separate file from your data. If there is a memo file associated with a data base, then the "memnam" field of the data base header will have the name of the memo field; the "memolen" and "memowid" will have the length and width of the memo field itself.

If a record in your data base has a memo associated with it, the "rptr" field in the header that occurs prior to each record will have a value. With this value, you can calculate the offset you need to go to in the memo file in order to read the memo itself. Given "rptr", the formula:

```
offset = ((rptr - 1) * 256) + 6
```

yields the offset itself.

The memo file consists of a 6 byte header, followed by 256 byte memo blocks. The header looks like this:

```
struct {
   unsigned memsig = 0x334D;  /* memo file signature */
   unsigned long firstdel;    /* first deleted memo block */
};
```

Memo blocks look like this:

```
struct {
   unsigned long nxtblk;      /* next block for this memo */
   char       memo[252];      /* memo text */
};
```

Thus, you read the memo file in 256 byte chunks. Take the value of "rptr" and calculate the offset into the memo file. Go to that offset and read 256 bytes. The first four bytes will either point to the next 256 byte chunk you need to read or will be zero, indicating that there are no more blocks for this memo. The remaining 252 bytes are plain text.

Conclusion

Don't be scared away by the complexity of Clarion data base files. With this technical bulletin, you should be able to read most any data base, unless it is encrypted or compressed. Our format contains many parts, each serving a specific purpose. So go ahead, explore away!

