CLARION TECHNICAL BULLETIN

Bulletin #118

Clarion Index and Key Files

Overview

This bulletin explains the layout of Clarion index and key files.

Clarion Index and Key Files

While Technical Bulletin #117, "Clarion Data Files," covered the layout of Clarion Professional Developer's data files, it only touched on the subject of the index and key files that Professional Developer uses to access your data in a specific order. The purpose of this technical bulletin is to explain how Clarion's key and index files are laid out.

This technical bulletin assumes a knowledge of basic data structures and the C programming language. While you don't have to be a C guru to understand the material presented here, a passing knowledge helps, as C makes it easy to create many types of data structures.

As far as this discussion is concerned, there are NO internal differences between index and key files. Index files are simply key files that are not updated automatically by adding, deleting, or changing records in your data base.

Also, this bulletin will not explain how you create key files, only how to read them. Our method of managing B+ Trees is proprietary and quite complicated. In all fairness, we can only support products that create key files using our own code. When you see a non-Clarion product that creates and/or maintains Clarion key files, it means they have licensed our code for use in their product.

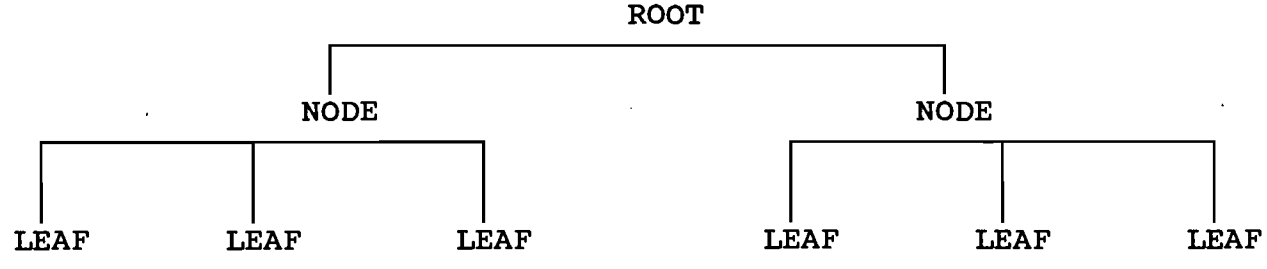
Let's Begin

Computers are very good at storing many types of information. However, in today's business world, just being able to store a lot of data isn't good enough; accessing this data quickly is of primary importance. It doesn't do you any good for your computer to store 1,000,000 records if it takes 30 minutes to find a specific record.

Clarion products use a method of indexing called "B+ Trees" (pronounced "Bee Plus Trees"). B+ Trees are a modification of another indexing method called "B-Trees," which were created by a man named R. Bayer in 1970. Let's start by seeing what a B-Tree is and how it works.

The Ubiquitous B-Tree (with apologies to Doug Comer)

A B-Tree is a data structure that, when drawn on paper, resembles an upside-down tree. B-Trees consist of a ROOT node; zero or more interior NODES, and zero or more LEAVES. A typical tree would look like this:



In actuality, all pieces of the tree are NODES, but they are given special names because of properties unique to each one. In any B-Tree, there is only one ROOT node. It is where the tree starts. LEAF nodes are

nodes that have no child nodes. All other nodes are called "interior" nodes, or just NODES.

Nodes contain pairs of key-pointer structures. The key will contain whatever data you've built the tree from and the pointer will either be a reference to another node or to data in your data base. You read the tree by starting at the root and looking at the key values. If the key you have is less than the first key in the root, you take the first branch. If the key you have is greater than or equal to the first key in the root, but less than the second, take the second branch, and so on. When you take a branch, continue with the same operation. When you finally reach the leaf level, you'll either find a match (meaning a record with the key you are looking for exists in the data base) or you won't. If you do find a match, the pointer in the leaf will tell you where in the data base the matching record is. The nice thing about B-Trees is that if you run into the "not found" condition, you are currently at the place in the tree where the key would be added if you wanted to add it to the tree. However, you can't just create a data structure that looks like this and call it a B-Tree. B-Trees have specific properties, several of which make the B-Tree as efficient as it is. For example, B-Trees are balanced trees, which means that no matter what data you add to the tree, it always takes the same number of accesses to retrieve a record.

Let's look at the properties of a B-Tree. A B-Tree is a B-Tree of "order" X if the following is true:

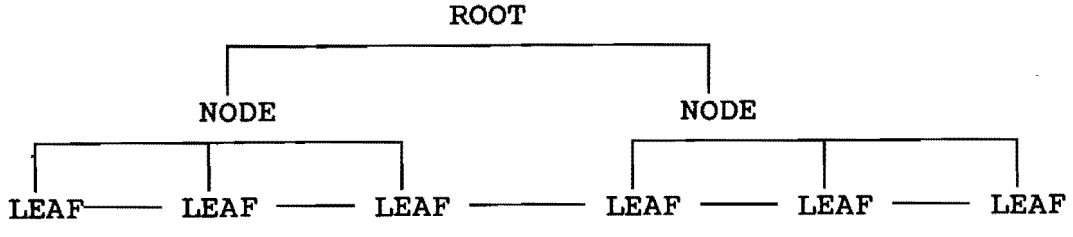
1) The ROOT has at least two children, unless the tree is empty, in which case the ROOT node has no children at all.

2) Each node has a maximum of X children, but each node IS AT LEAST half full (this does not apply to the ROOT or the LEAVES). This means that all interior nodes have at least X/2 children, but no more than X children. For example, if a B-Tree is of order 6, then all interior nodes will have at least 3 children, but no more than 6.

3) Each path in the tree from the ROOT to a particular LEAF is the same length.

Notice that this discussion keeps mentioning "child nodes." This is quite common in B-Trees. Nodes can have children. They also have parent nodes. You can carry this even farther with "grandparent" and "grandchild" nodes. All nodes (with the exception of the ROOT) have a parent. All nodes (with the exception of the LEAVES) have children. All nodes are linked together so you can traverse the tree. In a typical B-Tree, only the leaves actually point to data; the rest of the nodes point to other nodes.

Perhaps you are wondering what the difference between a B-Tree and a B+ Tree is. The main difference is that a B+ Tree has links to the left and right between the leaves. Thus, the previous diagram would change a little:



If you want to add a key to a tree, first find out where in the tree your new key should go; then insert the key into the proper place in the tree. However, this procedure is not as simple as it sounds. Remember, B+ trees have specific rules regarding their layout. You can't have more than "X" keys per node, where "X" is the order of the tree. Thus, if adding a key to a node causes it to overflow, you must then "split" the node into two parts. This means, for example, that adding a key to a leaf node might cause the node to overflow, thus making you split the leaf into two new leaves. This split of the leaves will cause a new entry to be made into the parent of the leaf. If this causes the parent to overflow, you must then split the parent. You can see that this could propagate up the entire tree. Indeed, some additions will cause splits all the way to the root node, sometimes causing a new root to be created.

This "cascading" effect can also happen during deletions. For example, if you delete a key in a node and this causes the node to have less than X/2 entries in it, you then try to combine the node with less than X/2 entries with some neighbor node that has X/2 entries. Let's say you do this and you have to combine two neighboring leaves. When you combine the leaves, you will remove an entry from the parent node of the leaves, since the parent will have pointers to both leaves. Again, let's say that this causes the parent node to have less than X/2 keys, thus causing the parent node to be combined with some other parent node. Like additions, deletions also can result in reorganization of the entire tree. But it is this reorganization that keeps the tree "balanced."

Clarion Index/Key Files

That's enough theory. Let's take a look at Clarion key files and see what they look like.

Clarion key files consist of a header and a sequence of zero or more nodes. Empty key files contain only a header. Once you add a key, nodes will be added to the key file. The key file itself tells you nothing about the fields that make up the keys. For that, you need to look at the data file.

The key file is read and written in blocks of 512 bytes. These 512-byte units are referred to as "nodes," except when talking about the header. If, for some reason, a node is not full, extra bytes are set to 0.

First, the header. Here is how the header is laid out:

```
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 number of end of file */
        unsigned long keybof;    /* record number of beg. of file */
        unsigned long unused;    /* first unused node of file */
        unsigned char keytyp;    /* type of key */
        unsigned char keynode;   /* number of keys per node */
        unsigned      keylen;    /* total length of key entry */
        unsigned      numlvls;   /* number of levels */
        char       cvoid[477];   /* reserved space */
};
```

The "cvoid" variable is used to fill the header out to a length of 512 bytes. The only variable in the header that is not a true numeric value is the "keytyp" variable. It is a bitmap that tells you information about the key itself. It is defined as follows:

```
struct {
        unsigned ftyp    : 4;    /* key or index? */
        unsigned dupsw   : 1;    /* duplicates allowed? */
        unsigned uprsw   : 1;    /* upper case only? */
        unsigned optsw   : 1;    /* optionals (blanks or zeroes) */
        unsigned locksw  : 1;    /* locked switch */
};
```

For those of you not familiar with C, this structure allows you to address individual bits in a variable. Since "keytyp" is contained in an 8-bit byte, the above structure is 8 bits in size. The number after the colon is the number of bits for a variable. So, if you look at "keytyp" as a sequence of 8 bits, 4 bits can be referred to as "ftyp," 1 bit for "dupsw," 1 bit for "uprsw," 1 bit for "optsw," and 1 bit for "locksw." The 1 bit variables can either have a value of 0 or 1; the 4 bit variable can have the values 00, 01, 10, or 11. Currently, only 00 and 01 are used.

Following the header are the nodes. A total of 13 bytes of each 512 byte node is allocated to a "node header," which contains links between all of the nodes. Node headers look like this:

```
struct {
        unsigned char keycnt;    /* number of keys in this node */
        unsigned long flink;     /* forward node pointer */
        unsigned long blink;     /* backwards node pointer */
        unsigned long ulink;     /* upwards node pointer */
};
```

After the node header comes the pointer/key combinations themselves. They look like this:

```
struct {
        unsigned long relrec;    /* record number/node number */
        char key[];              /* key value (key size - size of long)*/
};
```

The number of pointer/key combinations in a node is determined when the key file is initially created. You can't have less than 2 pointer/key entries per node; thus, if you have 512 bytes available and you allocate 13 bytes for a node header AND you have to give up 4 bytes for the pointer (relrec) in each pointer/key entry, that leaves you with a maximum key size of 245 bytes [(512 - 13 (header) - 8 (size of 2 longs)) / 2]. This is also the method that determines how many pointer/key combinations to put in a node.

Reading a key file is fairly simple. Once you have determined from the header how many pointer/key entries are in each node and how long each key is, you perform the following steps:

1) Start at the root node, which is located at offset "root" times 512.

2) Initialize a counter to keep track of what "level" you are at in the key file; set this counter to 1.

3) Look at the pointer/key structures until you find a key that is greater than or equal to the key you are looking for.

4) Once you have found this key, compare the "numlvls" variable to your counter. If they match, you are at the leaf level; the pointer is an actual record number in your data file; otherwise it is a node number.

5) Read the record if found in step 4.

6) If the pointer is an actual record number, you have found the record you need to look at in the data file. If not, go read the node the pointer refers to, increment your level counter, and go back to step 3.

When your level counter is equal to "numlvls," you are at the leaf level of the tree. If the key you are seeking is not at the leaf level, then your key is not in the key file. An example should help clear up any confusion. Let's look at a sample key file for a database of state abbreviations. The key is 2 characters long (STRING(2)) and is the standard 2 character state abbreviation. The data file contains the actual state name that corresponds to the 2 character abbreviation. To make this example use more than one node in a key file, some fictitious state abbreviations were added. These are the abbreviations AA, BB, CC, DD, etc. and 00, 11, 22, 33, 44, 55, 66, 77, 88, and 99.

First, here is the header:

```
00000: 03 00 00 00 57 00 00 00 03 00 00 00 01 00 00 00 |....W...........|
00010: 4D 00 00 00 57 00 00 00 00 00 00 00 60 53 01 06 |M...W.......`S..|
00020: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
.........................................
.........................................
.........................................
001F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Some of the header was omitted because it contains nothing but 00. Here is a breakdown of the fields in the header:

(0000) root — the "root" node of the tree (03 00 00 00); we have to reverse the order of the words in long values and also the order of bytes within words, due to the way Intel CPU's store data. Thus, this is the value 00 00 00 03. The root node is node number 3.

(0004) numkent — the number of key entries in this key file (57 00 00 00); this key file contains 57 keys, which just happens to be the number of records in the data file. This is only true if all keys must be unique. If duplicate keys are allowed, the number of keys may be different from the number of records.

(0008) numnode — the number of nodes in this key file (03 00 00 00); this key file contains a total of 3 nodes.

(000C) lastnod — the number of the last node in this key file (01 00 00 00); this is the "last" node in the file; however, the last node of the file may only contain part of the leaf level.

(0010) keyeof — the record number of the "last" record in the data file (4D 00 00 00); this is the record number that contains the key of "highest" value; in this case, it is record 4D hex (77 decimal), which contains the state abbreviation for state "ZZ."

(0014) keybof — the record number of the "first" record in the data file (57 00 00 00); this is the record number that contains the key of "lowest" value; in this case, it is record 57 hex (87 decimal), which contains the state abbreviation for state "00."

(0018) unused — the number of the first "unused" node in the key file (00 00 00 00); when a node is deleted, the deleted nodes are chained together so that all nodes in the key file will be in use before the key file grows in size. This key file has no deleted nodes, so this field has a value of 0.

(001C) keytyp — the type of key (60); remember, this is a bit map, so we have to break it down according to the structure mentioned above: 60 hex = 0110 0000; the bit map is flipped, so the "0000" is the "ftyp" value. Since it is 0, it means that this is a key file; if this were an index file, this field would have a value of 1. The other 4 bits (0110) correspond directly to the 1 bit fields above: dupsw = 0, uprsw = 1, optsw = 1, loksw = 0. This means no duplicates, key is all upper case, key is filled with spaces, and this file is not locked.

(001D) keynode — the number of keys per node (53); this is the maximum number of pointer/key combinations that a node will hold. 53 hex is 83 decimal; 83 times 2 (the length of each key) + 83 times 4 (the length of the "relptr" pointer with each key) + 13 (the size of the node header) = 511.

(001E) numcmps — the number of components in this key (01); this key has only one component.

(001F) keylen — the total length of each key entry (06 00); each pointer/key combination takes up 6 bytes (2 for the key itself, 4 for the pointer).

(0021) numlvls — the number of levels in the key file (02 00); this key file has a total of 2 levels.

The rest of the header is filled with 0. Next come the nodes. First, node 1:

```
00200: 53 00 00 00 00 02 00 00 00 03 00 00 00 51 00 00 |S............Q..|
00210: 00 34 34 52 00 00 00 35 35 53 00 00 00 36 36 54 |.44R...55S...66T|
00220: 00 00 00 37 37 55 00 00 00 38 38 56 00 00 00 39 |...77U...88V...9|
00230: 39 34 00 00 00 41 41 19 00 00 00 41 48 18 00 00 |94...AA....AK...|
00240: 00 41 4C 31 00 00 00 41 52 07 00 00 00 41 5A 35 |.AL1...AR....AZ5|
00250: 00 00 00 42 42 03 00 00 00 43 41 36 00 00 00 43 |...BB....CA6...C|
00260: 43 2F 00 00 00 43 4F 26 00 00 00 43 54 32 00 00 |C/...CO&...CT2..|
00270: 00 44 43 37 00 00 00 44 44 27 00 00 00 44 45 38 |.DC7...DD'...DE8|
00280: 00 00 00 45 45 39 00 00 00 46 46 01 00 00 00 46 |...EE9...FF....F|
00290: 4C 04 00 00 00 47 41 3A 00 00 00 47 47 3B 00 00 |L....GA:...GG;..|
002A0: 00 48 48 1A 00 00 00 48 49 2C 00 00 00 49 41 30 |.HH....HI,...IA0|
002B0: 00 00 00 49 44 3C 00 00 00 49 49 20 00 00 00 49 |...ID<...II ...I|
002C0: 4C 2B 00 00 00 49 4E 3D 00 00 00 4A 4A 3E 00 00 |L+...IN=...JJ>..|
002D0: 00 4B 4B 17 00 00 00 4B 53 29 00 00 00 4B 59 0F |.KK....KS)...KY.|
002E0: 00 00 00 4C 41 3F 00 00 00 4C 4C 1F 00 00 00 4D |...LA?...LL....M|
002F0: 41 1E 00 00 00 4D 44 12 00 00 00 4D 45 1C 00 00 |A....MD....ME...|
00300: 00 4D 49 40 00 00 00 4D 4D 21 00 00 00 4D 4E 23 |.MI@...MM!...MN#|
00310: 00 00 00 4D 4F 0E 00 00 00 4D 53 10 00 00 00 4D |...MO....MS....M|
00320: 54 05 00 00 00 4E 43 09 00 00 00 4E 44 33 00 00 |T....NC....ND3..|
00330: 00 4E 45 24 00 00 00 4E 48 22 00 00 00 4E 4A 15 |.NE$...NH"...NJ.|
00340: 00 00 00 4E 4D 41 00 00 00 4E 4E 1D 00 00 00 4E |...NMA...NN....N|
00350: 56 0C 00 00 00 4E 59 13 00 00 00 4F 48 2D 00 00 |V....NY....OH-..|
00360: 00 4F 4B 42 00 00 00 4F 4F 1B 00 00 00 4F 52 0B |.OKB...OO....OR.|
00370: 00 00 00 50 41 43 00 00 00 50 50 44 00 00 00 51 |...PAC...PPD...Q|
00380: 51 25 00 00 00 52 49 45 00 00 00 52 52 06 00 00 |Q%...RIE...RR...|
00390: 00 53 43 0A 00 00 00 53 44 46 00 00 00 53 53 02 |.SC....SDF...SS.|
003A0: 00 00 00 54 4E 47 00 00 00 54 54 16 00 00 00 54 |...TNG...TT....T|
003B0: 58 2E 00 00 00 55 54 48 00 00 00 55 55 11 00 00 |X....UTH...UU...|
003C0: 00 56 41 0D 00 00 00 56 54 49 00 00 00 56 56 14 |.VA....VTI...VV.|
003D0: 00 00 00 57 41 2A 00 00 00 57 49 28 00 00 00 57 |...WA*...WI(...W|
003E0: 56 4A 00 00 00 57 57 08 00 00 00 57 59 4B 00 00 |VJ...WW....WYK..|
003F0: 00 58 58 4C 00 00 00 59 59 4D 00 00 00 5A 5A 00 |.XXL...YYM...ZZ.|
```


Here is a breakdown of the header for node 1:

(0200) keycnt — the number of keys in this node (53); this is the total number of keys currently stored in this node. Hex 53 = 83 decimal.

(0201) flink — the forward link (00 00 00 00); this is the link to the next node on the same level in the tree.

(0205) blink — the backwards link (02 00 00 00); this is the link to the previous node on the same level of the tree; in this case, node 2 is on the same level of the tree.

(0209) ulink — the upwards link (03 00 00 00); this is the link to the parent node of this node; in this case, node 3 is the parent of node 1.

Following this header are the pointer/key combinations. Here are the first few in the node:

Offset	Pointer	Key	Actual Value in Dump
020D	51	44	(51 00 00 00) (34 34)
0213	52	55	(52 00 00 00) (35 35)
0219	53	66	(53 00 00 00) (36 36)
021F	54	77	(54 00 00 00) (37 37)

Next is node 2:

```
00400: 04 01 00 00 00 00 00 00 00 03 00 00 00 57 00 00 |.............W..|
00410: 00 30 30 4E 00 00 00 31 31 4F 00 00 00 32 32 50 |.00N...11O...22P|
00420: 00 00 00 33 33 00 00 00 00 00 00 00 00 00 00 00 |...33...........|
00430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
.........................................
.........................................
.........................................
005F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Here is a breakdown of node 2:

(0400) keycnt — (04) Node 2 only has 4 pointer/key entries.

(0401) flink — (01 00 00 00) Node 2 has a forward link to node 1.

(0405) blink — (00 00 00 00) Node 2 has no backward link.

(0409) ulink — (00 00 00 00) Node 2 has no upward link.

Since node 2 only has 4 pointer/key entries, they are listed in their entirety here:

Offset	Pointer	Key	Actual Value in Dump
040D	57	00	(57 00 00 00) (30 30)
0413	4E	11	(4E 00 00 00) (31 31)
0419	4F	22	(4F 00 00 00) (32 32)
041F	50	33	(50 00 00 00) (33 33)

And finally, node 3:

```
00600: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 |................|
00610: 00 33 33 01 00 00 00 5A 5A 00 00 00 00 00 00 00 00 |.33....ZZ.......|
00620: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00630: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00650: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00660: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
.........................................
.........................................
.........................................
007F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Here is a breakdown of node 3:

(0600) keycnt (02) Node 3 only has 2 pointer/key entries.

(0601) flink (00 00 00 00) Node 3 has no forward link.

(0605) blink (00 00 00 00) Node 3 has no backward link.

(0609) ulink (00 00 00 00) Node 3 has no upward link.

Since node 3 only has 2 pointer/key entries, they are listed in their entirety here:

Offset	Pointer	Key	Actual Value in Dump
060D	02	33	(02 00 00 00) (33 33)
0613	01	ZZ	(01 00 00 00) (5A 5A)

If you visualize this key file as a tree, then node 3 is the root node and nodes 1 and 2 are leaves. Since this key file is not very big, it has no interior nodes.

So, how would we go about finding a key in this key file? Let's go through the steps to find the key "FL."

1) Read the header of the key file; this tells us that the key file has 2 levels, a maximum of 53 pointer/key structures per node, and each pointer/key structure takes up 6 bytes in a node.

2) Set our level counter to 1 and read the root node, which happens to be node 3. Node 3 has 2 pointer/key entries.

3) After skipping over the links in the node header, we come to the first pointer/key structure. It has a key value of "33," which is less than the key value we are looking for. Skip to the next pointer/key entry.

4) The next pointer/key structure has a key value of "ZZ," which is greater than the key we are looking for, so we follow the pointer associated with "ZZ." It has a pointer value of 01.

5) Since our level counter is 1 and we know that our key file has 2 levels, the pointer value is a pointer to another node and not a data record. Go read node 1.

6) Increment our level counter to 2. Since the key file only has two levels, we know that we are now on the leaf level with node 1.

7) Start looking at pointer/key structures until we find a matching key; if we find no matching key, then the key does not exist in the key file. In this case, we do find it at offset 028B. It has a pointer value of 1, which is the record number of our data file that has the name of the state with the abbreviation "FL."

This key file was fairly simple in nature, since the key is a string of only two characters. Let's take a look at the other data types and see how they get converted to key values.

Long — Long components are split into two "words," each 16 bits long. The order of these bytes are reversed; then the order of the words within the long are reversed. Example: If you have a long of the form AABBCCDD, the value is split into AABB CCDD; then the individual bytes are reversed, giving you BBAA DDCC. After that, you reverse the order of the words, giving you DDCCBBAA. Finally, toggle the high order bit.

Short — Short components are also split into two pieces and flipped; the high order bit is also toggled. AABB would turn into BBAA.

String — Strings are stored as-is, except they are converted to upper case if the key is not case sensitive.

Picture — Same as strings.

Group — Same as strings

Byte — Stored unchanged.

Decimal — Decimal values are handled the following way:

if the high order bit of the decimal value is turned on then flip all the bits in each part of the decimal else toggle the high order bit. In C, it would look something like this:

```
/* comp[] is the decimal number */
if (comp[0] & 0x80) {
        for (x = 0; x < KEYDESC.elmlen; ++x) {
                comp[x] = ~comp[x];
        }
} else {
        comp[0] |= 0x80;
}
```

Real — Real values take up 8 bytes; whether or not the value is positive or negative, the order of the bytes is reversed. If the value of the last byte in the original real value is negative, then the high order bit of that byte is flipped.

In C, it looks like this:

```
char *cp1;              /* the original real value */
char *cp2;              /* the final real value */

cp1 = [address of real value to be converted];
cp1 += 7;               /* start at the end of the number */
if (*cp1 < 0) {         /* is the end byte negative? */
        for (x = 1; x <= 8; x++) {
                *cp2++ = ~*cp1--;
```



```
        }
} else {
        *cp2++ = *cp1-- ^ 0x80;
        for (x = 2; x <= 8; x++) {
                *cp2++ = ~*cp1--;
        }
}
```