

# CINFO Scanning

*Efficient method*

Tue, Mar 19, 2002

The CINFO system table is designed to house information about analog channels that is sparse; i.e., it is not needed by most channels. The main use of this table so far has been to keep fast digitizer parameters that apply to a few channels—those that have a fast digitizer hardware support. Its main user is therefore LOOPFTPM, which supports the Acnet FTPMAN protocol.

There is a weakness in the support for searching the CINFO table that should be improved. Searches of this table are supported via the following routine:

```
CINFOE *CINFOentry(short typ, short chan);
```

This routine returns a pointer to the CINFO entry that matches the given type and channel. The type specifies the kind of entry, say, whether the form that supports Swift or Quick digitizers. The channel is, of course, the analog channel number for which the entry applies. The format of an entry is:

```
typedef struct {
    char cinfo_siz;
    char cinfo_typ;
    short cinfo_chan;
    long cinfo_dat[1];
} CINFOE;
```

The size of an entry is variable, but it can only occupy space in the table in 8-byte multiples. The sizes now in use are 8 or 16 bytes. The structure shown above is of size 8 bytes. If it were longer, there would be more 32-bit elements in the `cinfo_dat` array. If the `cinfo_siz` field is zero, it is interpreted as if it specified 8 bytes.

A search of this table by `CINFOentry()` involves scanning through each entry of the table to find a match on both `cinfo_typ` and `cinfo_chan`. Along with an access to `cinfo_siz`, this requires 3 accesses per entry. This would be ok, except when access to the table is slow. For PowerPC-based systems, the CINFO table is kept in nonvolatile memory, and a single access to that non-cacheable memory costs one microsecond. When the new Linac system was installed, this table in each front-end was allocated 512 entries, allowing for considerable future expansion. But a single search of a 512-entry table that fails to find a match would likely require 1.5 ms; multiple searches that explore different entry types that might match the same channel would require even more time.

This note describes a method for performing faster searches. The essence of the method is to copy the contents of the table into ordinary dynamic memory, where searches can be done much faster. Since the current means of altering the table's contents are done via ordinary memory accesses, the system code has no convenient way to know when the table contents have been modified. This means that the dynamic memory copy of the table should be treated like a cache, so that the copy may have to be repeated whenever too much time has passed since the last time.

The cost of the copy operation can be minimized by recognizing that a long word access requires the same time as a byte or word access. If one had to copy a 512-entry CINFO table in a PowerPC system, it could be done in about 1 ms. If the table is not full, and if we make the assumption that it cannot have large gaps between valid entries, the time needed to copy the table can be much less. We might assume that if two successive 8-byte entries were found empty during the copy operation, we could consider that all valid entries had been copied. Only the first long word of each 8-byte entry need be checked for this situation. (All valid `cinfo_typ` fields are nonzero.)

This proposal to improve the efficiency of scanning the CINFO table hides the logic of the copy operation inside the `CINFOentry` routine. This means that either a fixed part of dynamic memory is allocated to hold the copied table contents, or some static memory accessible to `CINFOentry` is used. When `CINFOentry` is called, it must be able to know whether its copy is stale, so that it can determine whether the copy must be refreshed. This means it needs access to a time-stamp, which may simply be a copy of the system cycle counter that is incremented every 15 Hz cycle.

Since `CINFOentry` is a function, it needs to refer to some static memory to implement the above plan. The static memory can simply be a pointer to an allocated block of memory large enough to match the declared CINFO table size. The allocated block could contain a time stamp that was captured when the block was last updated when copying from the nonvolatile CINFO table. It could also contain the number of entries for each type of entry—or the offset to the last entry of each type—so the search can stop once it has seen the last entry of the type being sought.

The `CINFOentry` logic simply checks for its static memory pointer. If it is `NULL`, or if the time stamp contained within the existing block header shows that too much time (perhaps one minute?) has elapsed since the last time the allocated block was updated, an update must be performed first, before the search can be made. To perform an update, copy the nonvolatile entries into the allocated block. Copy 8 bytes at a time using a pair of long word accesses. If two successive 8-byte entries have zero for the first of the pair of long words, it means that there is at least a 16-byte hole. If we assume that no such holes are valid, copying can stop at that point, in order to minimize the number of nonvolatile memory accesses required to perform the copying. During the copying, notice can be made of each entry type. A pointer for each possible type can be saved in the header, so that upon completion of the copy operation, a count of the number of entries of each type can be retained for use during subsequent searches. The entry types in current use are limited:

<i>Type#</i>	<i>Meaning</i>
0	empty
1	Swift/Quicker digitizer
2	Quick digitizer
3	1 kHz digitizer
4	15 Hz data pool

If we allowed for types in the range 1–16, it may serve for quite some time.

Note that using a one minute time for updating does not necessarily mean that an update will be performed every minute. It can only take place when `CINFOentry` is called. In the Linac environment, however, it may be that `CINFOentry` is called often enough so that an update is required nearly every minute. The reason for choosing one minute is so that any change made to the CINFO table will be recognized by `CINFOentry` within one minute.

It is useful to organize a search around entry types, but it would not be useful to do it for channel numbers. In the case of the 1 kHz digitizer, only a single entry is used, identifying the base channel number of a sequence of 64 channels. The base channel is always `0x0100`, so that if a search were made for this type that specified channel `0x0113`, it would fail. The client code knows this, so given channel `0x0113`, it invokes `CINFOentry` specifying the base channel `0x0100`, knowing that if that entry is there, it implies that the same entry applies for all channels in the range `0x0100–0x013F`.

### **Implementation results**

A new version of the `CINFOentry` code was installed in a test node, and measurements were made with a CINFO table containing one 16-byte entry, and another one with forty-one 16-byte entries. The header of the CINFO copy structure is defined as follows:

```
typedef struct {
    int cycleCnt;           /* CYCLECNT at last update of CINFO copy */
    int n8Entries;         /* number of 8-byte entries in CINFO copy */
    int copyCnt;           /* number of times copy operation performed */
    short copyTimeMx;      /* max elapsed time for copy */
    short copyTime;        /* elapsed time for copy, us */
    int yrmodahr, mnsccyms; /* BCD time of last copy */
    int searchCnt;         /* number of searches */
    short searchTimeMx;    /* max time for search, us */
    short searchTime;      /* time for search, us */
    short typCnt[maxTypesCINFO]; /* count of entries of each type */
    CINFOE cinfo[maxEntriesCINFO]; /* copied entries from CINFO table */
} CINFOCopyType;
```

Included within the header are several diagnostics that help characterize the effectiveness of the new code. For the example of a single 16-byte entry, we have the following result:

```

M MEMORY DUMP      03/19/02 0841
051A:8F58F8 0000 08F8 0000 0004 cycleCnt  n8Entries
051A:8F5900 0000 0002 0010 000D copyCnt  copyTimeMx,copyTime
051A:8F5908 0203 1816 4334 0000 yrmodahr mnsccyms
051A:8F5910 0000 0006 0000 0000 searchCnt searchTimeMx,searchTime
051A:8F5918 0000 0001 0000 0000 typCnt[16]
051A:8F5920 0000 0000 0000 0000
051A:8F5928 0000 0000 0000 0000
051A:8F5930 0000 0000 0000 0000

```

The `n8Entries` shows the number of 8-byte quantities, including the 2 empty entries at the end of the copy. The time to execute the copy operation is  $13 \mu\text{s}$  in this case, with  $16 \mu\text{s}$  as a worst case (out of two attempts). The BCD time at which the last copy operation was performed can be used as a crude record of how long ago `CINFOentry` has been called, in the case of a node that seldom hears of such `FTPMAN` requests. The number of searches ever performed is shown, along with the worst case search time. In this case, the search time was less than  $1 \mu\text{s}$ .

Another example is for the case of 41 entries of 16 bytes each:

```

051A:8F58F8 000D 3008 0000 0054 cycleCnt  n8Entries
051A:8F5900 0000 0003 0105 0105 copyCnt  copyTimeMx,copyTime
051A:8F5908 0203 1908 4119 0000 yrmodahr mnsccyms
051A:8F5910 0000 0009 0003 0000 searchCnt searchTimeMx,searchTime
051A:8F5918 0000 0029 0000 0000 typCnt[16]
051A:8F5920 0000 0000 0000 0000
051A:8F5928 0000 0000 0000 0000
051A:8F5930 0000 0000 0000 0000

```

Here we see that the copy time was  $261 \mu\text{s}$ . Taking the access time to the `CINFO` table to be exactly  $1 \mu\text{s}$ , there would have been  $84 * 2 = 168$  accesses amounting to  $168 \mu\text{s}$  needed to perform the copy. The worst case search, here attempted by specifying a channel number that yielded no match, showed  $3 \mu\text{s}$  only. For this example, the former logic might require  $1.5 \text{ ms}$  for searching a 512-entry `CINFO` table in nonvolatile memory. In practice, it is common for more than one search to be done, each specifying a different type, so that one could imagine  $4 \text{ ms}$  or so required for `FTPMAN` to figure out all it needs to know about a specified channel. The new logic is therefore a substantial improvement over the old.

The pointer to the allocated `CINFOCopyType` structure is kept in low memory at offset `0x6AC`.

Note that the new `CINFOentry` routine returns a pointer to a matching `CINFO` entry, but it refers to an entry in the `CINFO` copy structure and not in the original `CINFO` table. This means it cannot be used to modify the contents of the `CINFO` table itself.