A NETWORK REGISTRY

November 1989

This is a set of services, written in C, enabling registration and retrieval of "items". It is tailored to suit the CAI Ethernet bridge board. An item is a station on a net, its name is its address. This said, we can forget of it.

Environment

A short is supposed to hold 16 bits; Word is typedef'd as unsigned short; Name as Word[3]. (The upper-case-initial convention for types is taken from the proposed IEEE "spanning tree" bridge standard.)

An item has a descriptor that is a struct called Descr, comprising, among other things, an entry ("member") called name and being a Name. item is an instance of an array of Descr's, its size has to be greater than the number of items to be possibly registered but must not exceed 32768. item[i] with i in [1,size-1] is either unused or holds the descriptor of a registered item. (It may make sense to have a simply-linked "free list" of unused i's, to get/drop space for descriptors; maybe unfree descriptors ought to be linked too.) item[0] must not contain a descriptor (it can be put to other uses).

There is a directory defined as Word dirtry[0x8000], to be zeroed at start.

(The size 32768 is in fact the only painful restriction. It is easily relieved in C, but at a high cost for the resulting object code on a 86-type processor.)

Usage

Search

lookup(i,d,n) searches for item named n. If it gives i=0, the item is not registered and d is an empty directory place fit to hold the item's descriptor number in case it will be inserted. If it gives i>0, the item has been registered and i is its descriptor number (d is its place in the directory; of no interest here).

Insertion

For a new item, lookup gives an empty place in d. Now select a free descriptor number (preferably the smallest unused), i, fill in the name (and possibly other things). Then putin(d,i) will do the rest, putting the descriptor number into the directory.

Deletion

putout(n) deletes item named n from the directory and returns i, its (has-been) descriptor number. (No harm is done if called for an unregistered item; i will be 0.) Then the descriptor may, but needn't, be cleared, returned to free, whatever.

Do not insert or delete within interrupt routines, or make it safe in the obvious way.

Code

```
#define hash(h,n) \
h = n[0] ^ n[1] ^ n[2]; if (h & 0x8000) h ^= 0x8001


#define next(k) \
k = (k - 1) & 0x7FFF


#define found(i,n) \
( item[i].name[2] == n[2] \
&& item[i].name[1] == n[1] \
&& item[i].name[0] == n[0] )


#define lookup(i,d,n) \
{ hash(d,n); while ((i = dirtry[d]) && !found(i,n)) next(d); }


#define putin(d,i) \
dirtry[d] = i


Word putout(Name n)
{
Word b, c, d, i;
  lookup(i,d,n);
  b = d;
  while (next(b), c = dirtry[b])
  {
    hash(c,item[c].name);
    if ((c - b) >= (d - b)) { dirtry[d] = dirtry[b]; d = b; }
  }
  dirtry[d] = 0;
  return(i);
}
```

Remarks

That hash algorithm is essentially Amdahl-Ershov with a twist. What really happens is this. An item name is hashed through the directory, with the directory containing (rather than the name itself) the number of the descriptor (holding the name). - As to the expected performance and its connection with directory saturation, cf. "Hash Cash", CAI, Budapest 1974, or extract the relevant reasoning and relations from Knuth, vol. 3. For short: there is no problem as long as the directory has not been filled to more than two thirds, inasmuch as finding a registered item is likely to take less than two comparisons, finding a place for a not yet registered item, less than five comparisons (so the method is fast provided we have not much more than 20-24000 registered items).

Instead of deleting items, they could be "marked as deleted" (using otherwise-unused patterns in the directory to show that); algorithmic changes are straightforward (a directory location that contains a "marked as deleted" counts as "occupied" but irrelevant when searching, as "empty" when inserting). This is faster than true deletion, and is preferable if the set of possible items is fairly unvaried (that is, the disappearing items and the newly appearing ones are more or less the same); otherwise, performance is quickly deteriorating and eventually obstructed. Of course, a hybrid policy is feasible: mark deletions for a while, then do a cleanup. Still, for the time being, I suggest true deletion; later on, we can reconsider.

By the way, if the possible names belong to a previously known fixed set that is not greater than the number of items we are prepared to handle simultaneously, the directory size can be chosen between some senseful limits, and the hash function can be adapted, so as to optimize upon searching.

For found, some compilers would accept "*item[i].name == *n" and generate a better object code. Unfortunately, this is not standard.

It is worthwhile to write part of the code directly in assembler language; candidates are: the whole of lookup, or at least hash and found. When doing so (e.g., on a 16-bit processor), some small (but efficient) improvements offer themselves that cannot be indicated in C yet will be absolutely obvious while coding.

A Variant: One-Word Names

Now Name is to be typedef'd as Word. - With Ethernet, the least significant bit of an address has a special meaning not relevant in a bridged net, so such an address is but 15 bits. Therefore no hashing is needed, every possible name having its own place.

```
#define lookup(i,d,n) \
{ d = n>>1; i = dirtry[d]; }


#define putin(d,i) \
dirtry[d] = i


Word putout(Name n)
{
Word d, i;
  lookup(i,d,n);
  dirtry[d] = 0;
  return(i);
}
```