8 Map Library

The map tool is Scout's central hash-table manager. It provides three kinds of operations: a mutating operation (insertion and removal of an element), a lookup operation (lookup and membership test), and an iterator (visit all elements in a hash-table). See the Map Library Design Notes [6] for a description of how maps are implemented in Scout.

8.1 Type Definitions

`Map` *Type*

An abstract type that represents a hash table. Users access this table through the operations defined below.

`MapBinding` *Type*

An abstract type that is used as a handle to efficiently delete a hash-table entry.

`MapWalk` *Type*

This is an abstract type that holds all the state needed to visit the entries in a map.

`MapEl` *Type*

When traversing the elements in a map, the value returned by each call to `mapWalkNext` (see below) is a pointer to a map element structure, which is defined as follows:

```
typedef struct MapEl {
    ...private fields...
    AnyType      value;
    union {
        struct {
            size_t keySize;
            char   key[1];
        } var;
        struct {
            char key[1];
        } fixed;
    } u;
} * MapEl;
```

If the map uses a fixed key size, the key that the map element represents is stored in member `u.fixed.key`. Otherwise, the key is stored in `u.var.key` and the key size is stored in `u.var.keySize`. Note that even though the `key` members are declared to be just one byte long, they really are long enough to hold the entire key. In either case, the value that the key maps to is stored in member `value`.

8.2 Operations

8.2.1 Initialization

`void` **`mapInit`** `(void)` *Function*

This function is called once at system startup time by the Scout infrastructure before any maps are created. It initializes the map library.

8.2.2 Creating and Deleting Maps

`Map` **`mapCreate`** `(size_t` *n*`, ssize_t` *s*`)` *Function*

Creates a map with *n* hash-table buckets and a hash-table key size of *s* bytes. It is advisable to make *n* an integer power of two that is about twice as big as the number of expected entries in the hash table. The maximum number of buckets is currently limited to 1024. Note that hash-tables never overflow (as long as there is enough memory available). But to avoid degrading lookup performance, it is best to keep the ratio of the number of entries to the number of buckets below 0.5. If all hash-table keys have the same size, then *s* specifies that size. Otherwise, the value `-1` should be passed to request the creation of a variable key size hash-table.

`void` **`mapDelete`** `(Map` *m*`)` *Function*

Deletes map *m*. First, all remaining entries are unbound and freed, then the memory of the map itself is freed.

8.2.3 Binding and Resolving Ids

`MapBinding` **`mapBind`** `(Map` *m*`, const void *` *k*`, AnyType` *v*`)` *Function*

Creates a hash-table entry in map *m* that maps key *k* to value *v*. Argument *k* must be a pointer to a key that is `mapKeySize(`*m*`)` bytes long. This function may only be used for maps with a fixed key size. For maps with variable sized keys, use `mapVarBind` instead. On success, a map binding is returned. This is a handle that can be used in a call to `mapRemoveBinding` to quickly remove the entry. A call to this function fails if an entry for key *k* exists already and the value *v* does not match the value stored in the existing entry. In such a case, `NULL` is returned and `tSelf→errno` is set to `EENTRYEXISTS`. If there is an existing entry for *k* with a value that matches *v*, the binding for the old entry is returned (there is no reference counting for map entries).

Note that a common error in using this function is to forget to initialize every byte in key *k*. For example, compilers often "pad out" C structures to satisfy alignment requirements. To make sure that such padding is initialized, it is advisable to make a call of the form `memset(`*k*`, 0, mapKeySize(`*m*`))` before filling in the structure members. In general, it is best to avoid using structures as hash-table keys. A similar problem exists for argument *v*. To avoid problems, it is best to initialize *v* either as a pointer argument (*v*`.p`) or as a long integer (*v*`.l`). It would be wrong to initialize *v* as a short integer, for example.

`long` **`mapUnbind`** `(Map` *m*`, const void *` *k*`)` *Function*

Removes the entry in map *m* for *k*. Argument *k* must be a pointer to a key that is `mapKeySize(m)` bytes long. This function may only be used for maps with a fixed key size. The call succeeds and returns `0` if an entry for key *k* can be found. Otherwise, `-1` is returned and `tSelf→errno` is set to `ENOSUCHENTRY`.

`long` **`mapRemoveBinding`** `(Map m, MapBinding b)` *Function*

Removes the entry in map *m* that has map binding *b*. Semantically, this is equivalent to the corresponding call to `mapUnbind` but this function is faster and works for both fix sized and variable sized keys.

`long` **`mapResolve`** `(Map m, const void * k, size_t s, AnyType * v)` *Function*

Resolves the value that key *k* maps to in map *m*. If there is an entry for key *k*, its value is returned in `*v` and `0` is returned. Otherwise, `-1` is returned and `tSelf→errno` is set to `ENOSUCHENTRY`. This function may only be used for maps with a fixed key size. For maps with variable sized keys, use `mapVarResolve` instead. Argument *s* gives the key size of *k* and must be identical to `mapKeySize(m)`. Technically the *s* argument is redundant. However, if *s* is a small constant, the operation often executes significantly faster than in the general case. As long as the key *k* is appropriately aligned, it is safe to assume that this optimization takes places for integer types of all possible sizes (char, short, long) as well as for keys of the size of a pointer.

Note that a common error in using this function is to forget to initialize every byte in key *k*. For example, compilers often "pad out" C structures to satisfy alignment requirements. To make sure that such padding is initialized, it is advisable to make a call of the form `memset(k, 0, mapKeySize(m))` before filling in the structure members. In general, it is best to avoid using structures as hash-table keys.

`MapBinding` **`mapVarBind`** `(Map m, const void * k, size_t s, AnyType v)` *Function*

Like `mapBind` but for maps that use variable sized keys. The size of key *k* is passed in *s*.

`long` **`mapVarResolve`** `(Map m, const void * k, size_t s, AnyType * v)` *Function*

Like `mapResolve` but for maps that use variable sized keys. The size of key *k* is passed in *s*.

8.2.4 Walking a Map

Visiting the entries in a map consists of three steps: (1) the map walk context is initialized with `mapWalkInit`, (2) `mapWalkNext` is invoked until it returns `NULL`, and (3) `mapWalkDone` is called to free up the state in the map walk context.

`void` **`mapWalkInit`** `(MapWalk w, Map m)` *Function*

Initializes the map walk context *w* in preparation for visiting the entries in map *m*.

`MapEl` **`mapWalkNext`** `(MapWalk w)` *Function*

Returns the next entry in the map for which *w* was initialized. If there are no more entries, `NULL` is returned.

`void` **`mapWalkDone`** `(MapWalk w)` *Function*

Frees the map walk context *w*. There must be exactly one call to `mapWalkDone` for each call to `mapWalkInit`.

8.2.5 Miscellaneous

`size_t` **`mapKeySize`** `(Map` *m*`)` *Function*

Returns the key size with which map *m* was created, or `-1` in the case of a map that uses variable sized keys.