SymTab Manual

SymTab Manual

Version 1.0

SymTab Manual

Introduction Introduction

Thank you for having purchased SymTab. SymTab is a set of functions and data structures that provide a symbol table facility. SymTab is useful as a component of

- editors that need a facility for naming things.
- Macro processors.
- Compilers and interpreters of languages that have a stack based procedure calling mechanism. SymTab will support most conventional languages with little customization. More specifically, SymTab is a good for languages that either don't support local variables or do support local variables and meet the following criterion: The language is compatible with a stack based procedure calling mechanism. The lifetimes of local variables are tied either to procedures or other well defined parts of a program. When the part of a program is entered to which a local variable is tied, its lifetime begins. When that part of the program is exited, its lifetime ends.

This manual is divided into two sections: a slightly theoretical overview of symbol tables and a reference section. Even if you already know everything about symbol tables, we recommend that you skim through the overview to gain familiarity with the terminology used in this manual.

The distribution diskette contains:

symtab.c	This file contains all of the symbol table functions.
symtab.h	A header file for symtab.c. Most customizations to SymTab can be made by editing this file.
symtab.prn	This document.

symtab.c symtab.h symtab.c symtab.prn

If you have any comments or questions about SymTab please phone us at 415-680-1964 or write to us at

GeoMaker Software
P.O. Box 273124
Concord, CA
94527-3124

SymTab Manual

Chapter 1 C
hapter 1

SymTab Overview S
ymTab Overview

A symbol table is a collection of names paired with other information that we will refer to as the name's value. We will refer to the data structure that describes a single name−value pair as a binding. All names in a symbol table must either be unique or distinguishable as unique when taken with other information in the symbol table. What we refer to as a value need not consist only of numeric or string information, but will typically be a data structure that also contains type and other information.

A symbol table is also a group of functions for manipulating the collection of bindings. The most fundamental group of such functions consists of

- A function to add bindings.
- A function that takes a name, finds the like named binding and returns the value from the binding. We will refer to this function as lookup.
- A function that when given a name will remove the like named binding.

There are two different ways a binding can refer to a value: A binding can contain a value or it can point to a value. Bindings that contain values provide faster access to the values and simplify memory management. But, if the values vary greatly in size, bindings that point to values provide better memory utilization. The semantics of some languages require that bindings point to values so that the same value can be shared by multiple bindings.

Symbol tables often exist as part of a language compiler or interpreter. The data structure used by SymTab to represent
1
the symbols or variables being compiled or interpreted is the binding.

1. We use the terms symbol and variable interchangeably in this document.

SymTab Manual

Most languages support a kind of variable that has an indefinite lifetime. These exist from the time they are defined until the termination of the program or until their lifetime is explicitly ended by an executable statement. For the purposes of SymTab, we will refer to this type of variable as a global variable. The features required of a symbol table to support global variables are few. Only the three previously mentioned functions are required. The remove function may be unnecessary for languages that have no a way to end the lifetime of global variables before the termination of a program.

Many languages also support one or more kinds of variable that have a definite and well defined lifetime. We will refer to this type of variable as a local variable. The rules that languages have for the lifetime of local variables differ, but one property of local variables is almost universal: The lives of multiple local variables can and do begin at the same time and end at the same time. So frequently multiple local variables have identical lifetimes. To facilitate the management of variables with identical lifetimes, SymTab has a data structure called a scope.

A scope is just a list of bindings. The way in which you would want to use scopes in your application will depend on the semantics of the language that you are compiling or interpreting. SymTab can be configured to work with scopes in a number of different ways.

There is a common, but not universal, property of local variables: When the lifetime of one set of variables overlaps the lifetime of another set of variables, one set has a shorter lifetime than the other. The lifetime of the shorter lived set is wholly contained within the lifetime of the longer lived set. We will refer to this property as the last born first dead property. For the purposes of SymTab, we say the scope that represents the shorter lived set of variables is nested in the scope that represents the longer lived set of variables.

For many applications, such as C and PASCAL compilers, a good way to manage nested scopes is on a stack. Stacks follow a last in first out rule, similar to the last born first dead property. SymTab can be configured to provide a stack discipline for nested scopes. When SymTab is configured this way, the scope with the shortest lifetime is always at the top of the stack. The global scope is always at the bottom of the stack. The add and remove functions operate only on one scope. That scope is normally the top of the stack. The lookup function operates on multiple scopes. Lookup starts its search at what is normally the topmost scope and then proceeds down toward the bottom of

SymTab Manual

the stack until the search is successful or all scopes in the stack have been searched.

For other applications, a simple stack is not sufficient. For these, SymTab can be configured to provide a tree−like data structure called a cactus stack. A cactus stack is a collection of one of more simple stacks which share a common base. The SymTab implementation of a cactus stack consists of a tree. Each leaf of the tree is considered to be the top of one of the stacks in the collection. The stacks all share the root of the tree and possibly some internal nodes with each other as a common base. The root of the tree is always considered to be the global scope. Note that a simple stack is special case of a tree in which each node has no more than one child.

An example of an application that requires a tree discipline is a C language interpreter. The root of the tree (the global scope) would contain the external symbols. The immediate children of the root node would contain outer level static variables and other symbols that are accessible only within the file in which they exist. The children of these nodes would correspond to the functions in the file and contain the local variables of those functions. Depending on the architecture of the interpreter, it might be necessary to have a node for each invocation of each function.

SymTab keeps bindings in linked lists. If a list is long it can become rather time consuming to search. SymTab has a feature that can significantly reduce the time spent in searching these linked lists. This feature takes advantage of a property that most programs exhibit called locality of reference. In our context, locality of reference means that the more recently a symbol has been referenced, the more likely it is to be referenced again. To take advantage of locality of reference, SymTab can be configured to make all the linked lists of bindings beself organizing. This means that when a lookup operation finds a binding it is moved to the beginning of the linked list in which it was found. If there is locality of reference then a self organizing list makes the bindings most likely to be searched for become the easiest to find.

Some languages, such as C, require support for multiple name spaces. This means that multiple symbols can have the same name and be distinguished by their context. For example, in C it is possible for a function to contain a variable, a structure tag and a label with the same name. One way to support multiple name spaces is to have a separate symbol table for each name space, but that would be inefficient. Having multiple symbol tables would result in duplication of data structures and multiply the bookkeeping overhead for nested scopes. SymTab can be configured so that multiple

name spaces are maintained in the same symbol table. When SymTab is so configured, every function that takes a name as one of its parameters also takes a name space parameter.

SymTab Manual

Chapter 2 C
hapter 2

SymTab Reference
SymTab Reference

Most customizations necessary to taylor SymTab to your application can be performed by editing the file symbol.h.
symbol.h
This chapter first describes the data structures of SymTab and what you need to know to customize them. Then this chapter describes the functions available in symtab.c.
symtab.c
Finally some performance tuning options are discussed.

2.1 Data Structures 2.1 Data Structures

It is usually important that symbol table operations happen quickly. For this reason, the collection of name−value pairs and related data structures are kept in main storage. The combination of data structures needed varies with the application. Configuring the data structures that SymTab uses is accomplished by editing typedefs and #defines in symtab.h.
symtab.h

2.1.1 Values 2.1.1 Values

One customization should always be performed: Specify the data type that is to be used to contain the values of bindings. This is done by replacing the typedef of sym_value_type as void * with one that defines sym_value_type as a data type appropriate for your application. Usually this type is some sort of struct.

For many applications it makes more sense for bindings to contain, rather than point to, their associated values. This is the default configuration. If you want bindings to point to their associated value, then uncomment the #define for SYM_POINTS_TO_VALUE.

You may need to do some sort of bookkeeping when a value is added to or removed from the symbol table. This is a common requirement for language interpreters that perform automatic memory management. Hooks into the SymTab functions are provided for this purpose.

When a value becomes referenced by a binding, either during the creation of a new binding or through the replacement of a value already referenced by a binding, SYM_ADD_VALUE_BOOKKEEPING is called. The default definition provided for SYM_ADD_VALUE_BOOKKEEPING is an empty #define. By providing your own definition you can do any necessary bookkeeping when values enter the symbol table. SYM_ADD_VALUE_BOOKKEEPING is passed a single parameter which is a sym_value_type *. No return value is expected.

Similarly, when a binding ceases to refer to a value, either during the removal of a binding or when an old value is replaced with a new value, SYM_REMOVE_VALUE_BOOKKEEPING is called. The default definition provided for SYM_REMOVE_VALUE_BOOKKEEPING is an empty #define. By providing your own definition you can do any necessary bookkeeping when values leave the symbol table. SYM_REMOVE_VALUE_BOOKKEEPING is passed a single parameter which is a sym_value_type *. No return value is expected.

2.1.2 Names

2.1.2 Names

The type of the name field of a binding is sym_name_type. The default typedef for sym_name_type is const char *. If you change this then you should also make a corresponding change in the #define for sym_extract_name so that it still produces some sort of char *.

By default, SymTab uses the library function strcmp to determine equality of names. To alter this you should change the #define for SYM_COMPARE_NAME_FUNCTION. Whatever the definition, it should take two parameters that are both of type sym_name_type and return zero if they are to be considered equal or non-zero if they are to be considered non-equal.

If you redefine sym_name_type to not be a char * you should modify the #define of sym_extract_name so that it still produces a char * from a sym_name_type. If that is not feasible, you should:

1. Leave the definition of sym_extract_name as is.

2. Modify the #define of SYM_COMPARE_NAME_FUNCTION (see preceding paragraph) so that it can make practical use of your definition of sym_name_type.

3. In symtab.c change the type of the global variable symtab.c
   my_name to sym_extract_name.

4. Modify the function hash_bucket so that it can work with your definition of sym_name_type.

SymTab Manual

You may need to do some sort of bookkeeping when a name is added to or removed from the symbol table. Hooks into the SymTab functions are provided for this purpose.

When a name is added to the symbol table, which happens during the creation of a new binding, SYM_ADD_NAME_BOOKKEEPING is called. The default definition for SYM_ADD_NAME_BOOKKEEPING is an empty #define. By providing your own definition you can do any necessary bookkeeping when names enter the symbol table. SYM_ADD_NAME_BOOKKEEPING is passed a single parameter which is a sym_name_type *. No return value is expected.

Similarly, when a name leaves the symbol table, which happens during the removal of a binding, SYM_REMOVE_NAME_BOOKKEEPING is called. The default definition for SYM_REMOVE_NAME_BOOKKEEPING is an empty #define. By providing your own definition you can do any necessary bookkeeping when names leave the symbol table. SYM_REMOVE_NAME_BOOKKEEPING is passed a single parameter which is a sym_name_type *. No return value is expected.

2.1.3 Global Scope 2.1.3 Global Scope

It is very common for programs to have a large number of global symbols. Because of this, SymTab uses a more sophisticated data structure to organize the global scope than the linked list it uses for nested scopes. Of the three basic operations (add, lookup and remove), by far the most frequently performed is lookup. Most names are looked up many times yet only added or removed once. Even the add and remove functions must do a lookup on the name. On the other hand, we are rarely interested in accessing the names in any particular order. Because of these characteristics, SymTab uses a chained hash table to store the bindings for

_____ ____ _____

global symbols. If you are unfamiliar with this scheme, it consists of an array of pointers to linked lists of bindings. We refer to these linked lists as hash buckets.

When searching for a binding, a hash function is first called to convert the name being searched for to an array index. Then the array is subscripted by the computed array index. The hash bucket pointed to by the selected array element is linearly searched for a binding with the given name (and name space if applicable).

The average time it will take to search for a binding in the hash table is proportionate to the average length of the hash buckets. The length of the hash buckets is inversely proportionate to the number of hash buckets. So there is a speed versus storage tradeoff in deciding on a good number of hash buckets for your purpose. To set the number of hash buckets change the value of the macro SYM_BUCKET_MAX in symtab.h. For best results, this number should be a prime

symtab.h

number. For your convenience, a selection of prime numbers is provided in a comment above the definition of SYM_BUCKET_MAX.

2.1.4 Nested Scopes

2.1.4 Nested Scopes

To configure SymTab to support nested scopes, uncomment the #define for SYM_DEEP_BINDING.

Nested scopes are usually organized into a stack. The maximum size of the stack is determined by the macro SCOPE_STACK_DEPTH which is shipped with a value of 100. While that is a sufficient value for most compilers, a larger value may be needed for interpreters. This is because compilers only concern themselves with lexically nested scopes whereas interpreters of some languages need to maintain dynamically nested scopes that result from run time procedure calls. For such an interpreter, 100 may be too small a number.

Some interpreters have more complex requirements for the management of scopes than can easily be accommodated by a simple stack. Many interpreters are required to maintain the local variables for multiple functions or procedures at the same time. For this and other complications, SymTab supports the alternate organization of scopes into a cactus stack. When SymTab is configured this way, the cactus stack is implemented using a tree for its storage structure. Every scope contains a pointer to its parent scope. To configure SymTab for a cactus stack, comment out the #define of SYM_SCOPE_STACK_DEPTH.

Most SymTab functions operate on or relative to the scope that is considered, at the time of the function call, to be the current scope. The current scope and functions for manipulation scopes are described in 2.2.4.

2.2 Functions

2.2 Functions

SymTab functions return one of the following values:

SYMTAB_NO_MEMORY
: Unable to allocate memory for a new binding.

SYMTAB_OK
: Normal return. All is well.

SYMTAB_DUPLICATE
: Name is already in the symbol table.

SYMTAB_NOT_FOUND
Name was not found.

SYMTAB_STACK_OVERFLOW
Attempt to exceed maximum depth of the scope stack.

SYMTAB_STACK_UNDERFLOW
Attempt to pop back to a more global scope than the global scope.

Most of the functions described below take a name as one of its parameters. If SYM_MULTIPLE_NAME_SPACES is defined in symtab.h then these functions also take a name space
symtab.h
parameter. The name space can be identified by any integer value, but lookup operations will succeed only if both name and name space match.

2.2.1 Adding to the Symbol Table 2.2.1 Adding to the Symbol Table

SymTab provides two flavors of add operation:

```
#ifdef SYM_MULTIPLE_NAME_SPACES
   int sym_add(sym_name_type name,
               sym_value_type *value,
               unsigned name_space);
#else
   int sym_add(sym_name_type name,
               sym_value_type *value);
#endif
```

The function sym_add adds a new binding with the given name and value to the current scope. SYMTAB_OK is returned if the new binding was successfully added. SYMTAB_DUPLICATE is returned if sym_add did not create the new binding because its name would duplicate a name (and name space if applicable) in the current scope (see 2.2.4). SYMTAB_NO_MEMORY is returned if sym_add failed because it was unable to allocate memory for a new binding. A typical use of sym_add is to process declarations for languages that have declarations.

```
#ifdef SYM_MULTIPLE_NAME_SPACES
   int sym_enter(sym_name_type  name,
                 sym_value_type *value,
                 unsigned name_space);
#else
   int sym_enter(sym_name_type  name,
                 sym_value_type *value);
#endif
```

The function sym_enter conditionally adds a new binding with the given name and value to the current scope. SYMTAB_OK is

returned if the new binding was successfully added. SYMTAB_DUPLICATE is returned if the name duplicates a name (and name space if applicable) in the current scope (see 2.2.4). In this case, the value of the existing binding is replaced with the given value. SYMTAB_NO_MEMORY is returned if sym_enter failed because it was unable to allocate memory for a new binding. Typical uses for sym_enter are to change the value of an existing symbol and to create symbols that require no declaration.

2.2.2 Looking Up Values

2.2.2 Looking Up Values

The function for that performs the lookup operation is:

```
   int sym_lookup(sym_name_type  name,
#ifdef SYM_POINTS_TO_VALUE
                  sym_value_type **value
#else
                  sym_value_type *value
#endif
#ifdef SYM_MULTIPLE_NAME_SPACES
                 ,unsigned name_space
#endif
                                        );
```

First, an explanation of the parameters of sym_lookup: The first parameter is the name of the symbol to be looked up. The second parameter is a pointer to a location that is to be set to the value of the found symbol. In the case that SYM_POINTS_TO_VALUE is defined, the value of a symbol is considered to be a pointer. The third parameter of sym_lookup is the name space of the symbol and is expected only if SYM_MULTIPLE_NAME_SPACES is defined.

When sym_lookup is called, it searches the current scope for the given name (and name space if applicable). If SymTab is configured to support nested scopes, then other scopes on the stack below the current scope are searched.

If the search is successful, sym_lookup sets the contents of its second parameter to the value found and returns SYMTAB_OK. If the search is not successful, then SYMTAB_NOT_FOUND is returned.

2.2.3 Removing Bindings

2.2.3 Removing Bindings

```
#ifdef SYM_MULTIPLE_NAME_SPACES
   int sym_remove(sym_name_type  name,
                  unsigned name_space);
#else
   int sym_remove(sym_name_type  name);
#endif
```

SymTab Manual

The function sym_remove removes a binding with the given name (and name space if applicable) from the current scope. If successful, sym_remove frees the storage occupied by the binding and returns SYMTAB_OK. If not successful, SYMTAB_NOT_FOUND is returned.

2.2.4 Managing Nested Scopes 2.2.4 Managing Nested Scopes

As mentioned previously, SymTab provides a facility to manage nested scopes. This facility exists only if SYM_DEEP_BINDING is #defined in symtab.h. If symtab.h
SYM_DEEP_BINDING is not #defined then the functions and macros described in this section will not exist.

SymTab offers a choice of two different organizational disciplines for nested scopes. If SYM_SCOPE_STACK_DEPTH is defined in symtab.h then scopes will be organized into a symtab.h
stack. The maximum depth of the stack is the numeric value SYM_SCOPE_STACK_DEPTH is #defined to be. If SYM_SCOPE_STACK_DEPTH is not defined then scopes will be organized into a cactus stack.

2.2.4.1 Stack Organization of Scopes 2.2.4.1 Stack Organization of Scopes

When scopes are organized as a stack, the global scope is always considered to be at the base of the stack. To push nested scopes onto the stack use:

```
int sym_push_scope(void); /*afegeix apila un nou ambit.*/
```

The function sym_push_scope pushes a new scope that contains no bindings onto the top of the stack, makes that scope the current scope and returns SYMTAB_OK. If sym_push_scope fails because the stack is full, it returns SYMTAB_STACK_OVERFLOW.

To pop a scope off the stack use:

```
int sym_pop_scope(void); /*treu, desapila ambit cap pila.*/
```

The function sym_pop_scope removes all the bindings pointed at by the current scope, makes the previous scope the current scope and returns SYMTAB_STACK_OK unless

- the current scope is the global scope. In this situation, sym_pop_scope does nothing and returns SYMTAB_STACK_UNDERFLOW.
- the current scope is not the top of the stack. In this case sym_pop_scope does nothing and returns SYMTAB_NOT_TOP. The need for this check is explained below. If you want SymTab to be compiled without this /*Volem desapilar un ambit que no esta al cap de la pila.*/

```
      safety    feature,    uncomment    the   #define   for
      SYM_NO_CHECK_POP./*que no txequegi si es treu o no el*/
         /*cap de la pila.*/
```

The nature of your application may require an occasional violation of the stack discipline. SymTab provides mechanisms for bending the rules in an orderly manner. Firstly, there is a set of functions to directly access the global scope (see 2.2.5). There is also a way to return to a scope that is not necessarily the scope preceding the current scope. Returning to an arbitrary scope requires that we be able to identify the scope. There is a typedef sym_scope_id that is the type used to identify scopes. There is macro for getting the sym_scope_id of the current scope. The equivalent function prototype for this macro would be

```
sym_scope_id sym_get_scope(void);
```

The #define SYM_ROOT_SCOPE is the sym_scope_id of the global scope.

To permanently return to a given scope:

1. when a scope that you may later want to explicitly return to is the current scope, call sym_get_scope and remember the result.

2. To return to the remembered scope call sym_pop_scope until sym_get_scope returns the remembered sym_scope_id.

To temporarily return to a given scope, there is a function that can be called to set the current scope:

```
void sym_set_scope(sym_scope_id);
```

When SymTab is configured for stack organization of nested scopes, sym_set_scope will not exist unless you uncomment the #define for SYM_REQUIRE_SET_SCOPE.

So the procedure for temporarily returning to a given scope is:

1. when a scope that you may later want to temporarily return to is the current scope, call sym_get_scope and remember the result.

2. To return to that scope, call sym_get_scope and save the result so that you will be able to remember the current scope. Then call sym_set_scope to make the previous scope the current scope.

3. Call sym_set_scope to restore the previous current scope.

Do not use sym_set_scope to permanently return to a scope. While this is functionally equivalent to one or more calls to sym_pop_scope it does not free the storage consumed by the bindings contained in the popped scopes.

There is an alternative to remembering a previous scope. The macro sym_prev_scope can be used to determine the immediate predecessor of a scope. The equivalent function prototype for sym_prev_scope is

```
sym_scope_id sym_prev_scope(sym_scope_id);
```

There is no check made by sym_prev_scope for the bottom of the stack. If the argument of sym_prev_scope is equal to SYM_ROOT_SCOPE (the global scope) the result is not useful.

Under no circumstances should you attempt to make a scope that has been sym_poped the current scope.

2.2.4.2 Cactus Stack Organization of Scopes 2.2.4.2 Cactus Stack Organization of Scopes /*Saber d'on vols penjar l'ambit.*/

All scopes in a cactus stack have a unique identifier. The type of the identifier is defined by a typedef and is called sym_scope_id. The bottom of all the stacks in a cactus stack is the global scope. There is a #define that allows the global scope to be referred to as SYM_ROOT_SCOPE. There is always a current scope. The notion of a current scope is similar to a stack pointer. The current scope is usually, but not always the top of one of the stacks in the collection of stacks that make up the cactus stack. When the current scope is the top of one of the stacks that comprise the cactus stack, that stack is considered to be the current stack. There is a macro that returns the sym_node_id of the current scope. Its equivalent function prototype is

```
sym_scope_id sym_get_scope(void);
```

The function for adding a scope to the cactus stack is

```
int sym_push_scope(sym_scope_id);
```

Three things happen when sym_push_scope is called:

1. A new scope containing no bindings is created.
2. The new scope is made to point to the scope that is identified by the parameter of sym_push_scope.
3. The new scope becomes the current scope.

To push a new scope on to the top of the current stack one writes

```
sym_push_scope(sym_get_scope());
```

The value returned by sym_push_scope is SYMTAB_OK if all went well, or SYMTAB_NO_MEMORY if it was unable to allocate memory for a new scope.

The function for removing a scope from the cactus stack is

```
int sym_pop_scope(void);
```

When sym_pop_scope is called:

1. All of the bindings pointed at by the current scope are removed.
2. The scope pointed at by the current scope is made the top of stack.
3. The storage occupied by the current scope is freed.
4. The top of the current stack is made the current scope.
5. SYMTAB_OK is returned.

If sym_pop_scope is called when the current scope is the global scope, SYMTAB_STACK_UNDERFLOW is returned without any of the above actions having taken place.

If sym_pop_scope is called when the current scope is not the top of a stack, SYMTAB_NOT_TOP is returned without any of the above actions having taken place. SymTab can be compiled without this safety feature by uncommenting the #define for SYM_NO_CHECK_POP.

Since this is a cactus stack, there is a way to branch a new stack off of an existing scope. This is done by passing sym_push_scope the sym_scope_id of a scope that is not the top of a stack.

Something else you will eventually want to do is to switch stacks. The function to use for this purpose is

```
void sym_set_scope(sym_scope_id);
```

sym_set_scope makes the scope that is identified by its parameter the current scope. Passing sym_set_scope the sym_scope_id of a scope that is the top of a stack makes that stack the current stack.

The function sym_set_scope can also be used to make a node that is not the top of a stack the current scope. While the current scope is not the top of a stack, you should not call sym_pop_scope.

There is a macro for determining the ancestor of a given scope. Its equivalent function prototype is

```
sym_scope_id sym_prev_scope(sym_scope_id);
```

Under no circumstances should you attempt to make a scope that has been sym_poped the current scope.

2.2.5 Accessing the Global Scope

2.2.5 Accessing the Global Scope

Some languages have constructs that specifically access the global scope, independently of which scope is the current scope. SymTab provides a set of functions for this purpose. The default configuration of SymTab causes these functions to not be compiled. If you want to use these functions, uncomment the #define for SYM_REQUIRE_GLOBAL.

```
#ifdef SYM_MULTIPLE_NAME_SPACES
   int sym_global_add(sym_name_type name,
                      sym_value_type *value,
                      unsigned name_space);
#else
   int sym_global_add(sym_name_type name,
                      sym_value_type *value);
#endif
```

The function sym_global_add adds a new binding with the given name and value to the global scope. SYMTAB_OK is returned if the new binding was successfully added. SYMTAB_DUPLICATE is returned if sym_global_add did not create the new binding because its name would duplicate a name (and name space if applicable) in the global scope. SYMTAB_NO_MEMORY is returned if sym_global_add failed because it was unable to allocate memory for a new binding. A typical use of sym_global_add is to process declarations for languages that have semantically global declarations in a syntactically local context.

```
#ifdef SYM_MULTIPLE_NAME_SPACESsym_global_enter
   int sym_global_enter(sym_name_type name,
                        sym_value_type *value,
                        unsigned name_space);
#else
   int sym_global_enter(sym_name_type name,
                        sym_value_type *value);
#endif
```

The function sym_global_enter conditionally adds a new binding with the given name and value to the global scope. SYMTAB_OK is returned if the new binding was successfully added. SYMTAB_DUPLICATE is returned if the name duplicates a name (and name space if applicable) in the global scope. In this case, the value of the existing binding is replaced

with the given value. SYMTAB_NO_MEMORY is returned if sym_global_enter failed because it was unable to allocate memory for a new binding.

The function for that performs the global lookup operation is:

```
   int sym_global_lookup(sym_name_type  name,
#ifdef SYM_POINTS_TO_VALUE
                         sym_value_type **value
#else
                         sym_value_type *value
#endif
#ifdef SYM_MULTIPLE_NAME_SPACES
                        ,unsigned name_space
#endif
                                               );
```

First, an explanation of the parameters of sym_global_lookup: The first parameter is the name of the symbol to be looked up. The second parameter is a pointer to a location that is to be set to the value of the found symbol. In the case that SYM_POINTS_TO_VALUE is defined, the value of a symbol is considered to be a pointer. The third parameter of sym_global_lookup is the name space of the symbol and is expected only if SYM_MULTIPLE_NAME_SPACES is defined.

When sym_global_lookup is called, it searches the global scope for the given name. If SymTab is configured for nested scopes with stack discipline, then the other scopes on the stack are searched for the given name.

If the search is successful, sym_global_lookup sets the contents of its second parameter to the value found and returns SYMTAB_OK. If the search is not successful, then SYMTAB_NOT_FOUND is returned.

2.2.6 Removing Bindings

2.2.6 Removing Bindings

```
#ifdef SYM_MULTIPLE_NAME_SPACES
   int sym_global_remove(sym_name_type  name,
                         unsigned name_space);
#else
   int sym_global_remove(sym_name_type  name);
#endif
```

The function sym_global_scope removes a binding from the global scope. If successful, sym_global_remove returns SYMTAB_OK. If not successful, SYMTAB_NOT_FOUND is returned.

SymTab Manual

2.3 Performance Tuning 2.3 Performance Tuning

After you have incorporated SymTab into your application, you may want to tune performance. There are two types of adjustments that we suggest to SymTab for this purpose.

2.3.1 Self−organizing Lists 2.3.1 Self−organizing Lists

As mentioned in the previous chapter, all bindings are kept in linked lists. By default, these linked lists are not self−organizing. To make the linked lists self−organizing, uncomment the #define for SYM_SELF_ORGANIZING_LISTS./*optimitza codi*/

2.3.2 Tuning the Hash Bucket Search 2.3.2 Tuning the Hash Bucket Search

If your application makes frequent reference to the global scope, either directly or through sym_lookup, another area of optimization to consider is the function hash_bucket.

hash_bucket is internal to SymTab. Every time the global scope is accessed, hash_bucket is called. The purpose of hash_bucket is to compute the address a pointer in the hash table to a hash bucket. There are two ways in which hash_bucket can be optimized: one is by coding hash_bucket in assembler, but do not do this until you have investigated the algorithmic issues. The computation of a hash index should involve cheap operations. Perhaps more important, the nature of the computation should result in the lengths of buckets being roughly equal. If you want to determine the sizes of the hash buckets, you can call the function sym_histogram. A call to sym_histogram sends to standard output a listing of the size of each hash bucket.

If you want to use sym_histogram you must un−comment the #define of SYM_HISTOGRAM, otherwise sym_histogram will not be compiled.

```
/*per saber la distribucio per ambit global de la taula*/
/*taula de dispersio.*/
```

Finally, do not forget that if the hash buckets get too big, increasing the number of hash buckets by changing the definition of SYM_BUCKET_MAX will improve search time.

SymTab Manual

Index

Index

```
- 19 -
```

SymTab Manual

```
                sym_scope_id 13,          SYMTAB_NOT_FOUND
                     14                        10
                SYM_SCOPE_STACK_DEPTH     SYMTAB_OK 9
                     9, 12                SYMTAB_STACK_OVERFLOW
                SYM_SELF_ORGANIZING_LISTS      10
                     18                   SYMTAB_STACK_UNDERFLOW
                sym_set_scope 13,              10
                     14, 15
                sym_value_type 6          V
        V
                SYMTAB_DUPLICATE 9        value 2
                SYMTAB_NO_MEMORY 9
```

Contents

Contents