BITSET: Implementing Sets of Natural Numbers Using Packed Bits

Ola Ågren <Ola.Agren@cs.umu.se>

Version 2.0

October 2002

Abstract

BitSet is an implementation of sets of natural numbers in the language C, where the actual sets are packed in an array of bits. The API is universally usable wherever one or more sets of numbers is required, requiring no knowledge of the internal implementation details of the sets to be usable. This document shows both how to use it and also all required details of the implementation.

UMINF 02.10 ISSN 0348-0542

Contents

1 Introduction

This documentation describes BITSET, a library that implements sets of natural numbers[1]. Sets created by BITSET uses bits packed in an array in memory to hold the state for each possible number added in the set. This works best when sets of higher cardinality are required, rather then with small sets. It has been used for handling index files for a database implementation by the author.

The documentation is organized as follows: Section 2 describes the interface to BITSET, both formally as an abstract datatype and as the interface to be used by a C programmer. The next section (Section 3 on page 8) contains a Users' Guide on how to compile the library, access the library from other software, generate an updated version of the include file or regenerate the documentation. Section 4 on page 9 describes the background for BITSETs data representation and naming conventions. The last section (Section 5 on page 12) gives a short exposition on why things looks as they do in the source code.

2 Functionality

All standard functionalities for sets have been added, even though the names of the functions are somewhat non-standard (i.e., adding the suffix `Set` to the names). The sets comply with the following abstract datatype and the corresponding C function interfaces:

2.1 Abstract Datatype Set(int)

Name	**Parameter Type(s)**		**Resulting Type(s)**
`emptySet`	void	→	Set(int)
`empty2Set`	int	→	Set(int)
`disposeSet`	Set(int)	→	void
`copySet`	Set(int)	→	Set(int)
`insertSet`	int, Set(int)	→	Set(int)
`insertSameSet`	int, Set(int)	→	Set(int)
`removeSet`	int, Set(int)	→	Set(int)
`removeSameSet`	int, Set(int)	→	Set(int)
`singleSet`	int	→	Set(int)
`unionSet`	Set(int), Set(int)	→	Set(int)
`intersectionSet`	Set(int), Set(int)	→	Set(int)
`differenceSet`	Set(int), Set(int)	→	Set(int)
`isEmptySet`	Set(int)	→	Boolean
`memberOfSet`	int, Set(int)	→	Boolean
`chooseSet`	Set(int)	→	int
`equalSet`	Set(int), Set(int)	→	Boolean
`subsetSet`	Set(int), Set(int)	→	Boolean
`cardinalitySet`	Set(int)	→	int
`printSet`	Set(int)	→	void
`loadSet`	FILE	→	Set(int)
`saveSet`	FILE,Set(int)	→	int

[1] Using `unsigned long int`s as the type of the natural numbers.

2.2 Axioms

The following axioms hold true for the **Set(int)** sets (as long as there is sufficient memory available on the machine):

`insertSet`(X_1,`insertSet`(X_2,S)) ≡ `insertSet`(X_2,`insertSet`(X_1,S))
`insertSet`(X,`emptySet`()) ≡ `singleSet`(X)
`insertSameSet`(X,S) ≡ `insertSet`(X,S)[a]
`memberOfSet`(X,`emptySet`()) ≡ **false**
`memberOfSet`(X_1,`insertSet`(X_2,S)) ≡
    if X_1 = X_2
        then **true**
        else `memberOfSet`(X_1,S)
`chooseSet`(`emptySet`()) ≡ *ERROR*
`chooseSet`(`insertSet`(X,`emptySet`())) ≡ X[b]
`removeSet`(X,`emptySet`()) ≡ `emptySet`()
`removeSet`(X_1,`insertSet`(X_2,S)) ≡
    if X_1 = X_2
        then S
        else `insertSet`(X_2,`removeSet`(X_1,S))
`isEmptySet`(`emptySet`()) ≡ **true**
`isEmptySet`(`insertSet`(X,S)) ≡ **false**
`cardinalitySet`(`emptySet`()) ≡ **0**
`cardinalitySet`(`singleSet`(X)) ≡ **1**
`cardinalitySet`(`insertSet`(X,S)) ≥ `cardinalitySet`(S)

[a] This is true for the contents of the generated sets, but **not** for their position in memory.

[b] The `chooseSet` function is not deterministic when more than one element has been added to a set.

2.3 Functions of the Application Program Interface

These following C functions are available in the application program interface (API), with a short description of the functionality of each of the functions given as well.

A Boolean value **true** corresponds to an integer value of 1, and a Boolean value of `false` corresponds to an integer value of 0.

2.3.1 `unsigned long cardinalitySet(Set s)`

Returns the number of members in the set, i.e. $| s |$.

2.3.2 `unsigned long chooseSet(Set s)`

Return one member of the set (any one, this function is undeterministic).

2.3.3 `Set copySet(Set s)`

Returns a copy of the given set.

2.3.4 `Set differenceSet(Set s, Set t)`

Returns the set difference of s and t, i.e. $s \setminus t$.

2.3.5 `void disposeSet(Set s)`

Destructor for sets.

2.3.6 `Set emptySet(void)`

Returns the empty set, i.e. $\varnothing$.

2.3.7 `Set empty2Set(unsigned int i)`

Returns an empty set with a size of `i` $\times$ `SETINC` integers, mostly for internal use.

2.3.8 `int equalSet(Set s, Set t)`

Returns `true` if the sets *s* and *t* are equal and `false` otherwise.

2.3.9 `Set insertSameSet(unsigned long v, Set s)`

Add *v* to the set *s*.

2.3.10 `Set insertSet(unsigned long v, Set s)`

Add *v* to a copy of the set *s*.

2.3.11 `Set intersectionSet(Set s, Set t)`

Returns the intersection between *s* and *t*, i.e. $s \cap t$.

2.3.12 `int isEmptySet(Set s)`

Returns **true** if *s* is the empty set ($\varnothing$) and `false` otherwise.

2.3.13 `Set loadSet(char *n)`

Retrieves a set from a named file.

2.3.14 `int memberOfSet(unsigned long v, Set s)`

Returns `true` if *v* is a member of *s* and `false` otherwise.

2.3.15 `void printSet(Set s)`

Presents the content of a set on standard output.

2.3.16 `Set removeSameSet(unsigned long v, Set s)`

Removes *v* from the set *s*.

2.3.17 `Set removeSet(unsigned long v, Set s)`

Removes *v* from a copy of the set *s*.

2.3.18 `int saveSet(char *n, Set s)`

Saves the set *s* to a named file and sets (and returns) `errno` on error.

2.3.19 `Set singleSet(unsigned long v)`

Returns the singleton $\{v\}$.

2.3.20 `int subsetSet(Set s, Set t)`

Returns `true` if the set *s* is a subset of *t* and `false` otherwise.

2.3.21 `Set unionSet(Set s, Set t)`

Returns the union between *s* and *t*, i.e. $s \cup t$.

2.3.22 `insertSet`/`insertSameSet` **and** `removeSet`/`removeSameSet`

The difference between `insertSet` and `insertSameSet` (and also between `removeSet` and `removeSameSet`) is that the former creates a new set updated with the given value inserted (removed) and the latter updates and possibly moves/reallocates the given set. It must still be called in the same way, but `t = insertSameSet(1,t);` will give an updated `t` while `t = insertSet(1,t);` will yield `t` pointing to a new set while leaving the former set held by `t` in memory without any references. See Figure 1 for further explanation.

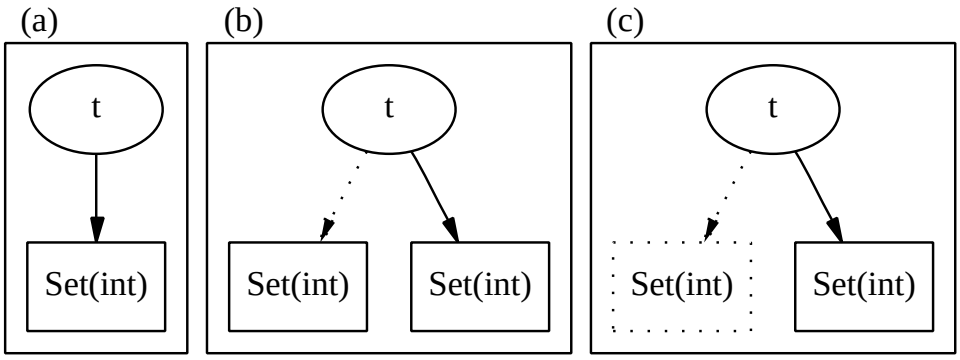

Figure 1: *The state of t and the corresponding sets: (a) before calling* `insertSet`/`insertSameSet`*, or after calling* `insertSameSet` *with a number that fits withind the current scope of the set; (b) after calling* `insertSet` *with* `t = insertSet(1,t);` *and (c) after calling* `insertSameSet` *with a number outside the current scope of the set, or first calling* `insertSet` *and then calling* `disposeSet` *on the former pointer.* The `dot` source code of this picture is given in Appendix C on page 14.

2.4 Accessibility Macros

Some extra functionalities are given in the `BitSetAcc.h` include file. These are mainly wrappers for the functions in the library, but they streamline the coding somewhat. F, s and t are names of a file, set and set, respectively.

2.4.1 `void FreeSet(Set s)`

An inline wrapper for `disposeSet` (see section 2.3.5 on page 5), makes sure that the set *s* points to `NULL` afterwards.
Interface: (s: Set(int)) → void

2.4.2 `int LoadSet(Set s, char *F)`

An inline wrapper for `loadSet` (see section 2.3.13 on page 5). Retrieves a set *s* from a named file, returns zero if the file could be loaded and non-zero otherwise (with `errno` set).
Interface: (s : Set(int),F : FILE) → int

2.4.3 `void differenceSameSet(Set s, Set t)`

An inline wrapper for `differenceSet` (see section 2.3.4 on page 4), will set *s* to point at the set difference (and remove the former value).
Interface: (s : Set(int), t : Set(int)) → void

2.4.4 `void intersectionSameSet(Set s, Set t)`

An inline wrapper for `intersectionSet` (see section 2.3.11 on page 5), will set *s* to point at the set intersection (and remove the former value).
Interface: (s : Set(int), t : Set(int)) → void

2.4.5 `void unionSameSet(Set s, Set t)`

An inline wrapper for `unionSet` (see section 2.3.21 on the page before), will set *s* to point at the set union (and remove the former value).
Interface: (s : Set(int), t : Set(int)) → void

The last three macros here use almost the same convention as `insertSameSet` and `removeSameSet`, but do not return anything. Instead they make sure that the variable *s* will point to the correct set after the call. The actual overhead of creating new sets and removing old ones is much lower for these three macros than for `insertSameSet` and `removeSameSet`, so the need for corresponding functions is negligible.

2.5 Format of Saved Bit Sets

There are functions available to save the contents of a set to file (`saveSet`) and to retrieve it again (`loadSet`). Each file contains the member(s) of the set in ascending order. Each line holds one member in hexadecimal format. There are two reasons for using this format rather than decimal:

1. It is faster to decode by the computer (since a byte normally corresponds to two hexadecimal digits); and
2. hexadecimal numbers take up less space than the corresponding decimal numbers, especially for larger numbers.

An example of such a file is given in Figure 2, in this case a set containing the decimal numbers 41, 100 and 204 is shown.

```
29
64
cc
```

Figure 2: A short example of a saved bit set.

3 Users' Guide

BitSet was originally created on a UNIX machine and compiled using `gcc` and `make`, but should be portable to any machine with an ANSI C compiler providing libraries and include files for assert, string and sys.

`Glib` has to be installed and be accessible on the system before compilation. Some of the routines that handle strings (like `g_strchug`) and the `MIN` and `MAX` macros are used.

3.1 Usage

The easiest way to get access to the functions from within a program is to include the `BitSetAcc.h` file[2] when compiling and adding the object file (`BitSet.o`) or corresponding archive/shared library file when linking. The program in Appendix D on page 15 interprets the two first arguments as file names of two saved bit sets, loads them, finds the intersection and presents it to the user.

3.2 Customization of the Makefile

The command line arguments to the C compiler have to be changed when moving the source package to a new machine. `GFLAGS` should be set to the output of `glib-config --cflags` and the "`-L`"-part of `LFLAGS` has to updated to the corresponding part from the output of running `glib-config --libs`.

[2]Do not forget to set up a `-I`-argument pointing to the correct include files. The `settype.h` file (as given in Appendix A on page 13) will automatically be included as well.

3.3 Customization of the Source Files

There is only one macro definition that might be interesting to change, and that is `SETINC`. It provides the minimum number of integers in an allocation block, i.e., each increase in the size of a set will be a multiple of these blocks. Moreover, each integer will hold the number of bits contained in a normal integer on the target machine.

Setting it to a large value is a waste of resources (a lot of unused memory) and will make all set operations take longer to perform. Setting it too low will result in increased number of allocations/reallocations. The latter is mainly a problem if the program is using functions that contain **Same** in the name, or the accessibility macros.

3.4 Compilation of an Object File

The command to compile the source code into an object file (after all changes required has been done according to sections 3.2 on the page before and 3.3) is `make`.

3.5 Generating a New BITSET Include File

Generation of a new BITSET include file is only required if changes have been made to the interfaces of the functions in the `BitSet.c` file or if additional functions have been added.

The actual generation of the `BitSet.h` file is performed when `make h` is executed. This operation requires that `cextract` is installed; see Section 3.6 for more details.

3.6 Generation of the Technical Documentation

The documentation should be included with the software package in PostScript and Portable Document Format, but it is possible to generate it from the sources. LaTeX[3], BiBTeX, `sed`, `egrep` and `cextract/cextdoc` must be installed in order to typeset the documentation from the source code. Almost all of the documentation is included within the `BitSet.c` source file, except those parts given in Appendices B on page 13 and C on page 14.

`cextract/cextdoc`[4] was written by Adam Bryant and extracts function comments, function signatures and optionally file comments from a set of C files. Furthermore, it can transform this information into pure text, `nroff`/`troff`/`groff` input or C header files.

The command for generating the documentation when everything has been set up correctly is `make doc`.

3.7 Generation Everything At Once

To generate everything at the same time (i.e., the include file, the object file and the documentation) use `make all`.

4 Implementation Rationale

The library is implemented using ANSI C to maximize portability. The native sizes of integers are used[5], and when it makes a difference (e.g., allocation units and access to the correct bit within an integer) it is to a large extent hidden by access macros.

[3] LaTeX2ε or later is required.

[4] Available at `http://dev.w3.org/cvsweb/Amaya/tools/cextract-1.7/`.

[5] Except for values within the sets that are `unsigned long` instead.

4.1 Choice of Data Representation

There are at least three obvious ways to implement a library for sets of natural numbers efficiently, each with strengths and weaknesses. The three are:

- **Linked lists of included members**, in size order:

  These are especially good when working with small sets (especially if they are sparse) or when checking for empty sets and cardinality. The downside is increased time for doing operations on larger sets, since it is necessary to step through each node in the lists before getting to the correct node (e.g., `memberOfSet`) or comparing the values of many individual nodes before finding the correct value (e.g., `unionSet`).

- **Arrays of cells** (e.g., byte/integer), where each cell contains a Boolean value that corresponds to the inclusion/exclusion of that value from the set:

  These are especially fast when it comes to adding/removing elements from a set, since changing the value of one single unit of the memory is required, and no calculation needs to be performed to find the correct cell where a value is stored. The main problem with this approach is that the memory requirement is high, especially if the largest value found in the set is very large.

- **Arrays of packed bits**, where each bit contains a Boolean value that corresponds to the inclusion/exclusion of that value from the set:

  These have their strength in memory usage, since only one bit is used for each value. The main drawback of this approach is that most modern computers lack bit operations which operates directly on memory, so bit operations on values residing in registers must be used instead. We have moreover found that the allocation units used affect the execution time within the library; making the allocation unit smaller results in increase in number of (re)allocations but fewer operations on "empty" memory cells and vice versa.

Implementations of the latter two types might be able to use SIMD instructions in the processor that operate on multiple units of memory directly, but that is up to the compiler. These instructions are available in most modern processors, e.g., MMX for the Intel x86 family of processors.

Arrays of cells would require too much memory and stepping through links in a large list would take too much time; thus the choice of packed bits. The resulting library is efficient even with large sets without using too much memory.

4.2 Conventions

A main problem was finding a suitable naming and calling convention for the functions in the library. Libraries should normally have either a common prefix or suffix to distinguish the functions from the functions in a user program.

4.2.1 Naming Conventions

The naming convention is derived from [JW00] but with a suffix of **Set** to reduce name space pollution. The reason for using a suffix is that `SetIsEmpty` feels more like a statement than a question about the state of the set; thus `isEmptySet` was chosen instead.

4.2.2 Calling Conventions

The calling convention is derived directly from [JW00], which means that data used to change a set is the first argument, with the the second the set which is operated on. The only interface where this does not hold true is the accessibility macro `LoadSet` (see Section 2.4.2 on page 7) where the set to update is given before the name of file to read.

The calling convention would probably be changed if a reimplementation of the library were to be performed later, so that the set operated on would be given first.

5 About the Documentation

This package is a test to see if it was indeed possible to keep both source code and documentation in one single file in order to minimize the conceptual distance between them. This is typical for literate programming [Knu84, Knu92] (including the webless version of the same, e.g., c-~~web~~ [Fox90]) and also POD for PERL and Javadoc [Fri95] for Java.

The main drawback of webbed literate programming is that information that logically belongs together (like the body of a function) is broken up in a number of pieces. Understanding what is happening in the code might force the user to skip back and forth between different (sub)sections of the documentation.

The main problem that we have found with c-~~web~~ is that it requires that the user write in TEX. This is a major drawback, since most users have migrated to LATEX. It also forces a certain document style on the user, e.g., having the front page and table of contents as the last page(s), which is not accepted by everyone. Moreover, all starts and ends of comments are retained in the final report, making it more difficult to read and giving it a "cluttered" appearance.

We wanted to be able to write the C code unimpeded by any *ad hoc* restrictions set up by the documentation system and *vice versa*. Using `cextract` together with `egrep` (to remove unwanted lines generated by `cextract`) and `sed` (to remove start and end of C comment markers) yields a clean source code for typesetting the documentation. Only a handful of lines in an enclosing LATEX file are required, see appendix B on the next page.

There is yet another difference between this approach and the literate programming approach, and that is that the actual implementation details of the functions (in this case the source code) are not shown. We feel that if users need to look at these details they should look at the source code and not the (possibly outdated) documentation.

References

[Fox90] Jim Fox. Webless Literate Programming. *TUGboat*, 11(4), November 1990.

[Fri95] Lisa Friendly. The Design of Distributed Hyperlinked Programming Documentation. In *Proceedings of the 1995 International Workshop on Hypermedia Design*, June 1995.

[JW00] Lars-Erik Janlert and Torbjörn Wiberg. *Datatyper och algoritmer*. Studentlitteratur, Lund, Sweden, second edition, 2000.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, (27), 1984.

[Knu92] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA, 1992.

A The `settype.h` Include File

```
#ifndef SETTYPE_H
#define SETTYPE_H
typedef struct  {
                int size;
                unsigned int data[0];
        } *Set;
#endif /* SETTYPE_H */
```

B The `BitSet.tex` LaTeX File

```
\documentclass[11pt,a4paper]{article}
\usepackage{isolatin1,amsmath,amssymb,varioref,moreverb,epsfig}
\usepackage[T1]{fontenc}

\addtolength{\textwidth}{20mm}
\addtolength{\hoffset}{-10mm}

\begin{document}
\input{BitSet2}

\newpage
\section{The {\tt settype.h} Include File} \label{settype.h}
\verbatimtabinput{settype.h}
\vspace*{1cm}

\section{The {\tt BitSet.tex} \LaTeX{} File} \label{BitSet.tex}
\verbatimtabinput{BitSet.tex}

\newpage
\section{The {\tt Same.dot} Dot(ty) File} \label{Same.dot}
This text is used to generate figure~\vref{insertSet}.
\verbatimtabinput{Same.dot}

\newpage
\section{A Sample of {\sc BitSet} Usage} \label{inter.c}
\verbatimtabinput{inter.c}

\end{document}
```

C The `Same.dot` Dot(ty) File

This text is used to generate figure 1 on page 6.

```
digraph G {
        subgraph clusterA {label="(a)";
                a [label="t"];
                b [label="Set(int)", shape=box];
                a -> b;
        }
        subgraph clusterB {label="(b)";
                c [label="t"];
                d [label="Set(int)", shape=box];
                e [label="Set(int)", shape=box];
                c -> d [style=dotted];
                c -> e;
        }
        subgraph clusterC {label="(c)";
                f [label="t"];
                g [label="Set(int)", shape=box, style=dotted];
                h [label="Set(int)", shape=box];
                f -> g [style=dotted];
                f -> h;
        }
}
```

D A Sample of BITSET Usage

```
#include <BitSetAcc.h>
#include <errno.h>
#include <stdio.h>

#define ERROR(c,s)      if ((c) != 0) { perror(s); exit(2); }

int main(int argc, char **argv)
{
        Set s = NULL, t = NULL, u = NULL;

        /* Check arguments (should use stat as well...) */
        if (argc != 3) {
                fprintf(stderr, "usage: %s file1 file2\n", argv[0]);
                exit(1);
        }

        /* Read the sets */
        errno = 0;
        ERROR(LoadSet(s,argv[1]),argv[1]);
        ERROR(LoadSet(t,argv[2]),argv[2]);

        /* Perform operation and present to user */
        u = intersectionSet(s, t);
        printSet(u);

        /* Delete all sets after use */
        FreeSet(s);
        FreeSet(t);
        FreeSet(u);
        return 0;
}
```