Bound-T timing analysis tool

# *find_marks*

# User Manual

TR-UM-004
Version 1
2009-04-04
Tidorum Ltd.


Tidorum Ltd
*www.tidorum.fi*
Tiirasaarentie 32
FI-00200 Helsinki
Finland

This document was written and is currently maintained at Tidorum Ltd by Niklas Holsti.

Copyright 2009 Tidorum Ltd.
This document can be copied and distributed freely, in any format or medium, provided that it is kept entire, with no deletions, insertions or changes, and that this copyright notice is included, prominently displayed, and made applicable to all copies.

| | |
|---|---|
| Document reference: | TR-UM-004 |
| Document issue: | Version 1 |
| Document issue date: | 2009-04-04 |
| *find_marks* version: | 1 |
| Last change included: | BT-CH-0169 |
| Web location: | *http://www.bound-t.com/find-marks-manual.pdf* |

Trademarks:
Bound-T is a trademark of Tidorum Ltd.

Credits:
This document was created with the free OpenOffice.org software, *http://www.openoffice.org*. We are grateful to Ada Core Technology (ACT) and the Free Software Foundation (FSF) for the GNAT Ada compiler that we use to compile *find_marks*.

The information in this document is believed to be complete and accurate when the document is issued. However, Tidorum Ltd. reserves the right to make future changes in the technical specifications of the product *find_marks* described here. For the most recent version of this document, please refer to the web-site *http://www.tidorum.fi/*. As *find_marks* is distributed under the GNU Public Licence, users may also change the program. This document describes the program as Tidorum provides it.

If you have comments or questions on this document or the product, they are welcome via electronic mail to the address *info@tidorum.fi* or via telephone, telefax, or ordinary mail to the address given below.

Please note that our office is located in the time-zone GMT + 2 hours, and office hours are 9:00 - 16:00 local time. In summer daylight savings time makes the local time equal GMT + 3 hours.

Cordially,

Tidorum Ltd.

Telephone: +358 (0) 40 563 9186
Fax: +358 (0) 42 563 9186
Web: *http://www.tidorum.fi/*
E-mail: *info@tidorum.fi*

Mail: Tiirasaarentie 32
FI-00200 Helsinki
Finland

# Contents



# Tables



# Figures

# 1 INTRODUCTION

## 1.1 Scope and purpose

***The Bound-T tool***

This document is the User Manual for the program called *find_marks*, a program provided by Tidorum Ltd under the GNU Public Licence (GPL). The *find_marks* program is an auxiliary program that is used to prepare input for the Bound-T program, also from Tidorum. Bound-T is a tool for developing real-time software – computer programs that must run fast enough, without fail. The main function of Bound-T is to compute an *upper bound* on the *worst-case execution time* of a program or subprogram. Bound-T does that by static analysis of the machine-code form of the program. For more information about Bound-T please refer to the User Guide, at *http://www.bound-t.com/user-guide.pdf*, or the Reference Manual, at *http://www.bound-t.com/ref-manual.pdf*.

***Assertions, and the need to identify program parts***

The task Bound-T tries to solve is generally impossible to automate fully. Finding out how quickly the target program will finish is harder than finding out if it will *ever* finish – the famously unsolvable "halting problem". For difficult target programs the user can control and support Bound-T's automatic analysis by writing *assertions*. An assertion is a statement about the target program that the user knows to be true and that bounds some crucial aspect of the program's behaviour, for example the maximum number of a times a certain loop is repeated.

Assertions are written in text files and expressed in the Bound-T assertion language as described in the Bound-T Assertion Language manual, *http://www.bound-t.com/assertion-lang.pdf*. Each assertion must somehow *identify* the part or parts of the target program to which the assertion applies. For example, an assertion on the maximum number of repetitions of a loop must identify *which* loop is meant. The Bound-T assertion language provides several ways to identify program parts. One of these ways is to use the *source-code position* of the part, which concretely means to identify the part by the source-line number and source-file name of some source-code line in or close to this part.

Consider, for example, the following C function *add_up*, where line numbers are shown on the left and only the start of the function is shown in detail:

```
33    ...
34    int add_up (int A[], int n)
35    {
36      int sum = 0, i;
37      for (i = 0; i < n; i++)
38      {
39        sum += A[i];
40        A[i] = 0;
41      }
...     ...
67    }
68    ...
```

The function contains a loop, *for* (*i* ...), for which Bound-T may not be able to find repetition bounds automatically, by analysis, which means that the user must supply the bounds by an asssertion.

If the user chooses (or is forced) to identify the loop by its source-code position, the assertion could be expressed as follows in the Bound-T assertion language:

```
subprogram "add_up"
  loop on line 37 repeats 21 times; end loop;
end "add_up";
```

The compiler and linker that generate the target program from the source code also create a mapping between source-code positions and machine-code addresses. This mapping is part of the debugging information in the executable target program (for example, an ELF file) and is accessible to Bound-T. When Bound-T analyses the subprogram *add_up* it creates the machine-code control-flow graph, which shows (among other things) the machine-code addresses of the instructions in the loop. If the compiler-generated source-to-object mapping is good enough, Bound-T can connect the machine-code address of the loop to "line 37" in the source code, and thus understand that this assertion should be applied to this loop.

### *Instability of line numbers, and countering it by offsets or marks*

So far so good, but what happens if the target program is modified by adding or removing some lines of code before the function *add_up*, in the same source-code file? Then all line numbers in *add_up* will change, so the line number in the assertion must also be changed. But updating assertion files in this way is cumbersome and error-prone.

The Bound-T assertion language offers two ways to solve this problem. One way is to use line-number *offsets* instead of absolute line numbers; this is explained in the Assertion Language manual and will not be discussed further here. The other way is to use *marks* embedded in the source code, which is where *find_marks* is useful and is our focus in this manual.

### *Example of source-code marks*

Consider the source code of *add_up* with one additional comment – a *mark*:

```
33    ...
34    int add_up (int A[], int n)
35    {
36      int sum = 0, i;
37      /**Mark line below "summer" */
38      for (i = 0; i < n; i++)
39      {
40        sum += A[i];
41        A[i] = 0;
42      }
...     ...
68    }
69    ...
```

With the help of the *find_marks* program Bound-T can now know that the marker-name "summer" means line 38 of this source-code file. The loop-bound assertion can therefore be written using this marker name instead of an actual line number:

```
subprogram "add_up"
  loop marked "summer" repeats 21 times; end loop;
end "add_up";
```

This assertion is *robust* against changes in the target program, as long as the "summer" mark remains on the line before the start of the loop.

***So what does find_marks do?***

The *find_marks* program reads source-code files, finds the mark lines, and outputs a table that shows the source-code position (file name, line number) of each mark. The table is stored in a text file, called a *mark-definition file*, that Bound-T reads together with the assertion files. Figure 1 below shows the inputs and outputs of *find_mark* and Bound-T. The area outlined by the dashed rectangle is the focus of this manual.

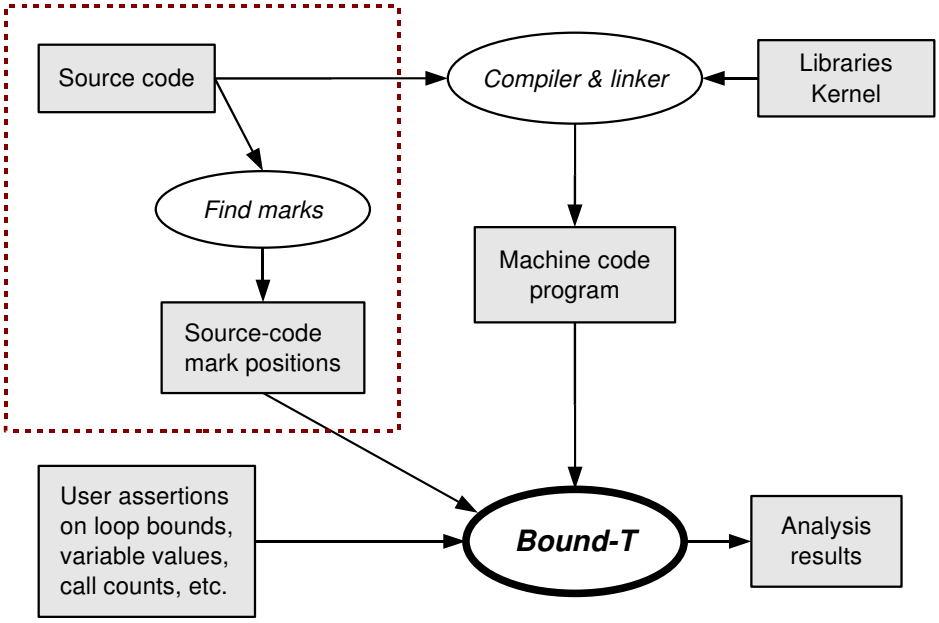


**Figure 1: Inputs, outputs, and context of *find_marks***

## 1.2 Overview of this document

This document is organised into chapters as follows:

- Chapter 2 shows how *find_marks* is used, that is, how to write a *find_marks* command and what the command-line options and arguments mean. This chapter also lists all warning messages and error messages from *find_marks*, with explanations and advice on solving the problems.
- Chapter 3 explains how to add marks to source-code files in any of the formats and programming languages that *find_marks* currently supports.
- Chapter 4 defines the format of the mark-definition files that *find_marks* produces and Bound-T consumes. This chapter is useful if you want to write a new mark-finder program from scratch: it specifies the format of the output that your program should produce.
- Chapter 5 describes the internal architecture of *find_marks* to help you extend or modify the program. The chapter ends with advice on how to add a new programming language (a new mark format) to *find_marks*.

# 2 USING FIND_MARKS

## 2.1 The *find_marks* command line

The *find_marks* program is executed from the command line and given a list of arguments that can contain any mixture of input (source-code) file names and options:

```
find_marks argument1 argument2 ...
```

The arguments that start with a hyphen '–' are interpreted as options. The other arguments are interpreted as the names of source-code files to be scanned for marks. The order of the arguments is meaningful: options apply to all following input files until overridden by new options. For example, the following command scans the file `libs.c` under the default options and the file `aux.txt` under the option `-c`:

```
find_marks libs.c -c aux.txt
```

***Storing the mark definitions in a file***

When *find_marks* finds a mark in an input file it writes the mark definition on the standard output channel. Thus, use the '>' redirection operator to store the output in a file, as in:

```
find_marks libs.c -c aux.txt >libs.marks
```

***Combining the results of several find_marks runs***

Mark definition files are “flat” text files with one mark definition per line and no headers or trailers. Thus you can use simple file concatenation to combine mark definitions from several executions of *find_marks*. For example, you can use the appending redirection operator '>>' if your command shell supports it:

```
find_marks libs.c >libs.marks

find_marks -c aux.txt >>libs.marks
```

The file `libs.marks` then contains both the marks from `libs.c` and those from `aux.txt`.

***Errors and warnings***

Errors and warnings from *find_marks* appear on the standard error channel. See Table 4 below.

## 2.2 Command-line options

Command-line options for *find_marks* fall into two groups: firstly, options significant to the general operation of *find_marks* and listed in Table 1 below; and secondly, options that define the type of source-code in the following input files, which defines the format of the marks in those files. The options in the second group are listed in Table 2 below.

All options are case-sensitive: the option *-ada* cannot be written *-ADA*.

**Table 1: General options**

| Option | Meaning and default value |
|---|---|
| At present there are no general options | |

**Table 2: Options for choosing input language**

| Option | Meaning and default value | |
|---|---|---|
| -ada | *Function* | Tells *find_marks* that the next input files contain Ada source code, until the next language-choosing option is found.<br>See section 3.3 for the Ada mark format. |
| | *Default* | See *-auto*. |
| -auto | *Function* | Makes *find_marks* choose the source language for the next input files based on the suffix ("file type") of the file name. |
| | *Default* | This is the default. The *-auto* option is useful to override a language chosen by earlier options. |
| -c | *Function* | Tells *find_marks* that the next input files contain C source code, until the next language-choosing option is found.<br>See section 3.4 for the C mark format. |
| | *Default* | See *-auto*. |
| -com=*comment-prefix*<br>-pre=*mark-prefix* | *Function* | Tells *find_marks* that the next input files, until the next language-choosing option is found, contain source code in some unspecified language but with marks defined by the given *comment-prefix* and *mark-prefix* strings and an optional mark suffix string (see the *-suf* option).<br>See section 3.5 for this mark format. |
| | *Default* | See *-auto*. |
| -suf=*mark-suffix* | *Function* | Augments the *-com* and *-pre* options by specifying a mark suffix string.<br>Note that this string is *not* a file-name suffix.<br>See section 3.5 for this mark format. |
| | *Default* | By default the mark suffix string is null, which means that mark lines have no ending suffix – the list of marker names is terminated only by the end of the mark line. |

***The -auto option***

The choice of an input language, for example with the option *-c* , is in effect for all following input files on the command line, until overridden by another choice. Take, for example, this command:

```
find_marks libs.c –ada aux.txt main.c >marks.txt
```

This command scans the file `libs.c` for markers in the C format, chosen by the file-name suffix (.c). The file `aux.txt` is scanned for markers in the Ada format, chosen by the preceding option *-ada*. However, this option is in effect also for the next file `main.c`, which is perhaps not intended. The command can be altered in several ways to use the C language format for `marks.c`:

- Move the file-name to an argument position that is not controlled by the *-ada* option:

```
find_marks libs.c main.c -ada aux.txt >marks.txt
```

- Add an option to choose the right language before the file-name:

```
find_marks libs.c -ada aux.txt -c main.c >marks.txt
```

- Use the option *-auto* to override the earlier option *-ada* and restore the default method for choosing the language based on the file-name suffix,:

```
find_marks libs.c -ada aux.txt -auto main.c >marks.txt
```

Table 3 below shows the mapping from file-name suffix to assumed programming language and mark format. Note that suffixes are compared in a *case-insensitive* way. For example, the suffix `ADB` is equivalent to `adb`.

**Table 3: Language from file-name suffix**

| File-name suffix | Language |
|---|---|
| `adb` | Ada |
| `c` | C |

## 2.3 Error and warning messages

Problems with the command-line options or the marks written in the input files can make *find_marks* issue a warning or error message on the standard error channel. The following table lists all these messages in alphabetical order, ignoring punctuation characters and letter case. For each message, the table explains the problem in more detail and may suggest possible reasons for the problem and specific solutions. Variable parts of the messages are shown in *italic* style and are not included in the alphabetical ordering.

**Table 4: Warning and error messages**

| Message | | Meaning and remedy |
|---|---|---|
| *file*:Cannot find this file. | *Reasons* | There seems to be no real file with this *file* name. |
| | *Action* | Correct the command line (file-name mistyped). |
| *file*:Cannot open this file. | *Reasons* | An attempt to open the source-code *file* named on the command line failed although the file seems to exist. Perhaps the file access permissions (“modes”) do not let you read the file. |
| | *Action* | Correct the file access permissions. |

| Message | | Meaning and remedy |
|---|---|---|
| *file*:Cannot read, perhaps not a text file. | *Reasons* | The *file* named on the command line could be opened, but could not be read as text. Perhaps the file is a non-text file such as a directory. |
| | *Action* | Name a readable source-code file. |
| *file*:No format chosen. | *Reasons* | No mark format (programming language) was chosen by options or by file-name suffix for this *file*. |
| | *Action* | Insert such an option on the command line before this *file* name. |
| *file*:*line*:No markable line above this line. | *Reasons* | The mark line in this *file*, on this *line* number, uses the **above** keyword to indicate that the marked line is the preceding markable line. But there is no earlier markable line in this *file*. |
| | *Action* | Correct the mark line. |
| *file*:*line*:No markable line for *n* pending "below" marks. | *Reasons* | There are *n* pending marks that are defined to mark the next markable line **below** the mark line, but this input *file* has been read to its end (at *line*) without finding any such markable line. |
| | *Action* | Correct the marks in this *file*. |
| Option "-*option*" not recognized. | *Reasons* | This command-line -*option* is not one that *find_marks* knows about. |
| | *Action* | Correct the command line (option mistyped). |
| *file*:*line*:"*word*" overrides earlier part keyword. | *Reasons* | The mark line in this *file*, on this *line* number, uses more than one keyword defining the part kind. |
| | *Action* | Correct the mark line. Use at most one "part" keyword. |
| *file*:*line*:"*word*" overrides earlier position keyword. | *Reasons* | The mark line in this *file*, on this *line* number, uses more than one keyword defining the position of the marked line. |
| | *Action* | Correct the mark line. Use at most one "position" keyword. |
| *file*:*line*:"*word*" overrides earlier relation keyword. | *Reasons* | The mark line in this *file*, on this *line* number, uses more than one keyword defining the relation of the marked part to the marked line. |
| | *Action* | Correct the mark line. Use at most one "relation" keyword. |
| Registering too many scanners (over *max*). | *Reasons* | The number of mark formats / programming languages registered exceeds the size *max* of the table. |
| | *Action* | Increase the constant *Marks.Scanners.Max_Number_Scanners*. |
| *file*:*line*:Source-file line is too long (over *max*) characters). | *Reasons* | This *line* in this *file* is longer (contains more characters) than the maximum *max* supported by *find_marks*. |
| | *Action* | Break the line into shorter parts, or increase the constant *Find_Marks.Max_Line_Length*. |

| Message | | Meaning and remedy |
|---|---|---|
| *file*:*line*:Too many “below” marks (over *max*) before the next markable line. | *Reasons* | When *find_marks* finds a mark line that uses the position keyword **below** it cannot emit the mark definition until the next markable line in this source *file* is found. The buffer has room for up to *max* pending marks, but the mark at this *line* no longer fits. |
| | *Action* | Increase the constant *Find_Marks.Max_Pending_Marks*. |

# 3 WRITING MARKS IN SOURCE CODE

## 3.1 Supported programming languages

Marks in source-code files are usually written as comments, because we usually want the compiler to ignore the marks. Each programming language has its own format for comments and so the format of marks also depends on the programming language. At the time of writing the *find_marks* program supports the programming languages listed in Table 5 below, as well as a generic kind of mark that is defined by the command-line options *-com*, *-pre*, *-suf*. The example given for the generic mark format assumes *-pre=';>>'* and *-suf='.'* .

**Table 5: Supported languages and mark formats**

| Language | Example mark line | Mark prefix | Mark suffix | Comment prefix |
|---|---|---|---|---|
| Ada | `---Mark line below "scan"` | `---Mark` | | `--` |
| C | `/**Mark line below "scan" */` | `/**Mark` | `*/` | `/*` |
| Generic | `;>> line below "scan" .` | Set by *-pre* | Set by *-suf* | Set by *-com* |

***Markable lines***

The purpose of a source-code mark is to give a name to a part of the machine code of the program to be analysed. This is done indirectly by giving a name to a source-code line that the compiler connects to the relevant instruction(s) in the machine code. It is therefore important to mark lines that *are* so connected, which means that the mark line itself – being a comment and not giving rise to any code – is not a good candidate, and it is probably better to use a nearby line that contains functional source code. For each programming language (each mark format) in *find_marks* some source-code lines are defined as *markable lines*.

The definition of a *markable line* depends on the chosen programming language, but for all languages currently supported a markable line is defined as any line that is not completely blank and does not start with the comment prefix defined in Table 5 above.

***Case sensitivity***

All text in a mark line – all prefixes, suffixes, keywords, and marker names – is processed in a case-sensitive way. You cannot write the keyword **call** as **CALL**.

***What follows***

While the initial and final parts of the mark lines differ according to the programming language, the core text of a mark line currently has the same form for all languages. The next section describes the core text form, and the later sections in this chapter discuss the mark formats for each supported input language.

## 3.2 Core text form

In all the currently supported mark formats a mark line consists of a prefix, a *core text* that actually defines the marks, and perhaps a suffix that terminates the mark text. The prefix and possible suffix depend on the chosen mark format (chosen programming language).

### *Property keywords followed by marker names*

The format of the core text is identical in all supported mark formats. The core text starts with zero or more *keywords* that define the properties of the mark. The property list is followed by a list of zero or more *marker names*.

The property keywords and the marker names are separated from each other and from the prefix and suffix (if present) by strings of whitespace characters. Here are some examples of such core texts:

| Keywords | Marker names |
|---|---|
| `call above` | `“rejection” “never”` |
| `loop spanning this line` | `“reduction”` |

### *Mark properties*

A mark has three properties that can be defined by keywords, or left undefined by default:

- The kind of program *part* that is marked: a **subprogram**, a **call**, or a **loop**.
- The *position* of the *marked line* relative to the mark line: **above**, **here**, or **below**.
- The *relation* of the part and the marked line: the part is **this** marked line, or it is the part **containing** the marked line, or the part **spanning** the marked line.

The *part* property is irrelevant to the operation of *find_marks* which simply conveys the value from the mark line in the source-code file to the mark definition in the output file. The significance, if any, of the part property depends on its use down-stream, when Bound-T reads and uses the mark definition file. This will be explained in the Bound-T assertion language manual.

The *position* property defines which source-code line number *find_marks* assigns to the mark, that is, which line is the *marked line*. For **here**, the marked line is the mark line itself. For **above**, it is the closest preceding markable line. For **below**, it is the closest following markable line.

The *relation* property, like the part property, is irrelevant to the operation of *find_marks*. The significance, if any, of the relation property depends on its use down-stream.

The keywords can be written in any order, but only one keyword for each property; you cannot override a property once defined. The keyword **line** can appear at any point and has no meaning; it is used just to make the text more grammatically pleasing. For example, **loop containing line above**.

### *Marker names*

A marker name is a string delimited by whitespace (thus, the string cannot itself contain whitespace characters). If you want to include commas (`,`) or quotes (`“`) in the name, you must enclose the name in quotes and write each quote in the name itself as two quotes. For example, the name `a”b` is written as `“a””b”`.

## 3.3 Marks in Ada code

***Mark format***

Mark lines in Ada source-code files have the following form:

- optional leading whitespace
- the mark prefix `---Mark`
- the core text (keywords and marker names).

***Examples***

```
---Mark “simple”

---Mark call here “anomaly” output
```

***Markable lines***

An Ada source-code line is considered a markable line if it contains some non-whitespace text and that text does not start with the Ada comment prefix `--` (two consecutive hyphens).

Since mark lines start with `---Mark` they are not themselves markable lines.

***File-name suffix***

The Ada format is assumed (under *-auto*) when the input file name has the suffix `adb` . This is the default suffix that the GNU Ada compiler GNAT uses for Ada subprograms and package bodies.

## 3.4 Marks in C code

***Mark format***

Mark lines in C source-code files have the following form:

- optional leading whitespace
- the mark prefix `/**Mark`
- the core text (keywords and marker names)
- an optional mark suffix that is `*/`

There must be some whitespace between the last marker name and the mark suffix. Any text on the line after the mark suffix is ignored; it can be compilable C code.

***Examples***

```
/**Mark simple */

      /**Mark loop spanning this line “polling”

/**Mark call here “anomaly” output */  report_error (1, “foo”);
```

***Markable lines***

A C source-code line is considered a markable line if contains some non-whitespace text and that text does not start with the C comment prefix `/*` .

Since mark lines start with `/**Mark` they are not themselves markable lines.

The C language allows multi-line or "block" commments in which the first line has the comment prefix `/*` , the last line has the comment suffix `*/` , and the lines in between can start with any text. The lines in between can thus be classified as markable lines although they are really comment lines. Avoid such block comments between a mark line and the intended marked line.

***File-name suffix***

The C format is assumed (under *-auto*) when the input file name has the suffix `c` .

## 3.5 Marks with selectable prefix and suffix strings

***The options and their meaning***

The command-line options *-com*, *-pre*, *-suf* control the operation of the generic mark format in *find_marks*, and also select this format for scanning the source files that are named after these options on the command line.

The *-com* option sets the comment prefix which controls the definition of markable lines: A source-code line is markable if it contains some non-whitespace text and that text does not start with the comment prefix.

The *-pre* option sets the mark prefix. A source-code line is a mark line if it contains some non-whitespace text and that text starts with the mark prefix.

The *-suf* option sets the mark suffix, which is optional. If the mark suffix is not a null string, the core text of a mark can be terminated by the appearance of the mark suffix as a whitespace-delimited non-whitespace string. The mark suffix is still optional – the core text can still be terminated by the end of the mark line, too.

Mark lines use the common core text format between the prefix and the suffix (if any): a list of keywords followed by a list of marker names, all separated by whitespace.

***Usage***

To define and choose the generic mark format for a source file, you must define both *-com* and *-pre* before the source-file name argument. Take for example the following command:

```
find_marks -com=';' lib.c -pre=';;;' sub.asm
```

Although *-com* is defined before the file-name `lib.c`, *-pre* is not and so the generic mark format is not yet chosen. The file `lib.c` is scanned using the mark format chosen by the file-name suffix, which means the C format.

However, before the file-name `sub.asm` both *-com* and *-pre* are defined. This completes the parameters for the generic format so `sub.asm` is scanned with the generic mark format using a single semicolon as the comment prefix and a triple semicolon as the mark prefix. The mark suffix is not defined so no suffix is used. To make this operation clearer, it is better to write the two options together, thus:

```
find_marks lib.c -com=';' -pre=';;;' sub.asm
```

After both *-com* and *-pre* are set once it is enough to use either of them to choose the generic mark format for the following files. For example:

```
find_marks -com=';' -pre=';;;' sub.asm -auto lib.c -pre='#' fuu.mac
```

This command scans `sub.asm` with the generic mark format using a single semicolon as the comment prefix and a triple semicolon as the mark prefix; then it scans `lib.c` with the C mark format (based on *-auto* and the `.c` suffix); and finally it scans `fuu.mac` with the generic mark format, still using a single semicolon as the comment prefix but now using a hash character as the mark prefix.

### *File-name suffix*

The generic mark format is never chosen based on a file-name suffix. It can only be chosen by the options *-com* and *-pre* .

# 4 MARK DEFINITION FILE FORMAT

## 4.1 Introduction

This chapter defines the format (syntax) and part of the meaning of mark definition files.

Section 4.2 defines the general format of the data fields and the general meaning of some of the data fields.

Section 4.3 explains how *find_marks* uses the format. Other tools that produce mark definition files may use the format differently.

This chapter does *not* explain how Bound-T uses mark definition files. That task is left to the Bound-T manuals, specifically the Assertion Language Manual.

## 4.2 Format of mark definitions

A mark definition file is a text file in which each line defines one mark. The file uses the CSV (comma-separated variable) style.

Each line in a mark definition file has five fields separated by commas (','). The last field is followed by end of line, not by a comma. Table 6 below describes the fields. The field names are defined only for use in this description and do not appear as such in the file.

**Table 6: Mark definition fields**

| Field | Name | Type | Content |
|---|---|---|---|
| 1 | *Marker* | String | The marker name. |
| 2 | *File* | String | The name of the source file that contains the marked line. |
| 3 | *Line* | Positive integer | The number of the marked source-code line in the file. |
| 4 | *Part* | Keyword | The kind of program part that is marked. |
| 5 | *Relation* | Keyword | The position and/or logical relation of the marked program part with respect to the marked source-code line. |

***String fields***

A field of type string contains a string of characters. If the string is not enclosed in quote marks (") it cannot contain commas or quote marks. A quote-enclosed string can contain commas and can contain quote marks if each such quote mark is written as two quotes ("").

***Integer fields***

A field of type integer contains a string of decimal digits (0123456789) which represent an unsigned integer number in the usual way (base 10).

The integer in the *Line* field must be positive. The first line in a source-code file is line number 1.

*Keyword fields*

A field of type keyword contains one of a finite set of strings; these strings are called keywords. Keywords cannot contain commas or quotes and are not enclosed in quotes. The set of keywords depends on the field. The following tables define the keywords, and perhaps some of their meaning, for each keyword field.

**Table 7: Keywords for the *Part* field**

| Keyword | Meaning |
|---|---|
| **any** | The kind of part that is marked is unspecified. |
| **subprogram** | A subprogram (procedure, function) is marked. |
| **loop** | A loop is marked. |
| **call** | A subprogram call is marked. |

**Table 8: Keywords for the *Relation* field**

| Keyword | Meaning |
|---|---|
| **any** | The position and relation of the marked part to the marked line are unspecified. |
| **here** | The marked part consists of or coincides with the marked line. |
| **above** | The marked part lies at or above (at equal or smaller line numbers than) the marked line, but in the same file. |
| **below** | The marked part lies at or below (at equal or larger line numbers than) the marked line, but in the same file. |
| **contain** | The marked part is an “extended” part, for example a loop, and some component (instruction) of this part is at the marked line. |
| **span** | The marked part is an “extended” part, for example a loop, and the number of the marked line falls in the range of source-line numbers connected to the marked part. |

*Examples*

For the *add_up* example in section 1.1 *find_marks* creates this mark definition line, assuming that the name of the source-code file is `subs.c` :

```
“summer”,subs.c,38,any,below
```

## 4.3 How *find_marks* uses the format

When *find_marks* finds a mark line (and its marked line) in an input file it creates a mark definition line as follows:

- The *Mark* field is simply the marker name, perhaps with enclosing quotes and doubled-quote encoding added.
- The *File* field is the source-file name exactly as it appears on the *find_marks* command line, perhaps with enclosing quotes and doubled-quote encoding added.

- The *Part* field contains the keyword that represents the “part” property of the mark, as described in section 3.2 (page 10).
- The *Relation* field contains a keyword that represents the combined “position” and “relation” properties of the mark. For details see the source code of *find_marks*, but the main point is that the mark-line keywords **containing** and **spanning** are translated to the *Relation*-field keywords **contain** and **span**, respectively.

If the mark line does not specify the kind of the marked part, or the position or relation of the marked line to the mark line, the keyword **any** is put in the *Part* or *Position* fields, respectively.

# 5 ARCHITECTURE OF FIND_MARKS

## 5.1 Modules

The *find_marks* program is written in Ada and consists of the Ada modules and Ada source-code files described in Table 9 below.

**Table 9: Modules in *find_marks***

| Module | Role | Source files |
|---|---|---|
| *Find_Marks* | Main procedure (main program). | `find_marks.adb` |
| *Marks* | Root package for processing marks and generating mark definition files. | `marks.ads`<br>`marks.adb` |
| *Marks.Formats* | Package that selects the mark formats to be included in a given version of *find_marks*. | `marks-formats.ads`<br>`marks-formats.adb` |
| *Marks.Scanners* | Package that defines the abstract root class for language-specific mark formats. | `marks-scanners.ads`<br>`marks-scanners.adb` |
| *Marks.Scanners.Ada* | Package that defines the Ada mark format. | `marks-scanners-ada.ads`<br>`marks-scanners-ada.adb` |
| *Marks.Scanners.C* | Package that defines the C mark format. | `marks-scanners-c.ads`<br>`marks-scanners-c.adb` |
| *Marks.Scanners.Fixed* | Package that defines a generic mark format parametrized by some prefix and suffix strings. | `marks-scanners-fixed.ads`<br>`marks-scanners-fixed.adb` |
| *Marks.Scanners.Opt* | Package that defines the mark format controlled by the command-line options *-com* , *-pre* , *-suf* . | `marks-scanners-opt.ads`<br>`marks-scanners-opt.adb` |

## 5.2 Classes and types

The only important class (or tagged type hierarchy in Ada terms) in *find_marks* is the class rooted at the abstract type *Marks.Scanners.Scanner_T*. An object in this class (necessarily of a derived type) represents one specific mark format. Such objects can be "registered" in the set of supported formats (*Marks.Scanners.Set*) and can then react to command-line options and to the suffixes of input-file names and, when chosen for an input file, are invoked to scan input lines to find marks in that file.

The derived types in this class are currently the following:

- *Marks.Scanners.C_Scanner_T*, which defines the C format of marks.
- *Marks.Scanners.Fixed.Scanner_T*, which defines a generic format parametrized by prefix and suffix strings.
- *Marks.Scanners.Opt.Opt_Scanner_T*, which is derived from *Fixed.Scanner_T* and implements the command-line options *-com* , *-pre* , *-suf* .

The Ada mark format is defined as an object of type *Marks.Scanners.Fixed.Scanner_T* in *Marks.Scanners.Ada*. It does not have a type of its own. The C scanner could also have been so defined, but it serves as an example of defining a mark format by a specific derived type.

## 5.3 Adding a new mark format

You can extend *find_marks* to support a new mark format (a new source-code language) in one of two ways:

- If the format can be described by fixed prefix and suffix strings, you can simply add an object of the type *Marks.Scanners.Fixed.Scanner_T* with those strings as component values. For an example, see *Marks.Scanners.Ada*.
- Otherwise, you must derive a new type from *Marks.Scanners.Scanner_T* (or from *Fixed.-Scanner_T*), write new format-specific operations to override the primitive operations that handle command-line options and file-name suffixes and scan source-code lines, and declare an object of this new type. For an example, see *Marks.Scanners.C*.

We recommend that you define a new package for your additions and name it *Marks.Scanners.<language>*, analogous to *Marks.Scanners.C*.

In both methods, remember to:

- Invoke *Marks.Scanners.Register* to register the object that defines the new format in the set of available mark formats ("scanners"). This is best done in the elaboration statement block at the end of your package body.
- In *Marks.Formats* add a "with" clause for the package that declares the object that defines the new format. This ensures that the package's statement block is executed at elaboration time (start-up) to register the new scanner object.

Tidorum will be glad to help you extend *find_marks*. Do not hesitate to ask us for advice or assistance!

Tidorum Ltd

Tiirasaarentie 32
FI-00200 Helsinki, Finland
*www.tidorum.fi*
Tel. +358 (0) 40 563 9186
Fax +358 (0) 42 563 9186
VAT FI 18688130