Bound-T time and stack analyser

Application Note

Renesas H8/300

TR-AN-H8300-001

Issue 1
2010-02-25

Tidorum Ltd.

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

This document was written at Tidorum Ltd. by Niklas Holsti.
The document is currently maintained by the same team.

Document reference:	TR-AN-H8300-001
Document issue:	Issue 1
Document issue date:	2010-02-25
Bound-T/H8/300 version:	4a2
Web location:	*http://www.bound-t.com/app_notes/an-h8-300.pdf*

Credits:
This document was created with the free OpenOffice.org software, *http://www.openoffice.org/*.

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 Bound-T described here. For the most recent version of this document, please refer to the web-site *http://www.bound-t.com/*.

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 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
E-mail:	*info@tidorum.fi*
Web:	*http://www.tidorum.fi/*
Mail:	Tiirasaarentie 32 FI-00200 Helsinki Finland

Credits

The Bound-T tool was first developed by Space Systems Finland Ltd. (*http://www.ssf.fi*) with support from the European Space Agency (ESA/ESTEC). Free software has played an important role; we are grateful to Ada Core Technology for the Gnat compiler, to William Pugh and his group at the University of Maryland for the *Omega* system, to Michel Berkelaar for the *lp-solve* program, to Mats Weber and EPFL-DI-LGL for Ada component libraries, and to Ted Dennison for the *OpenToken* package. Call-graphs and flow-graphs from Bound-T are displayed with the *dot* tool from AT&T Bell Laboratories.

Major parts of the H8/300 version of Bound-T were implemented by Samuel Petersson as part of his M.Sc. thesis at Mälardalen University, Sweden, with the support of the Advanced Software Technology Center (ASTEC) in Uppsala, Sweden (*www.astec.uu.se*) and of Vinnova, the Swedish Agency for Innovation Systems (*www.winnova.se*). Andreas Ermedahl supervised this thesis.

Contents

Tables

Document change log

Issue	*Section*	*Changes*
1	All	First issue.

1 INTRODUCTION

1.1 Purpose and scope

Bound-T is a tool for computing bounds on the worst-case execution time and stack usage of real-time programs; see references [1] and [2]. There are different versions of Bound-T for different target processors. This Application Note supplements the Bound-T User Guide [1] and Reference Manual [2] by giving additional information and advice on using Bound-T for one particular target processor, the processor architecture known as the Renesas H8/300 [4]. This processor was earlier known as the Hitachi H8/300.

The first goal of this document is to explain the additional command-line options and other controls that are specific to the H8/300 version of Bound-T. The second goal is to explain the sort of H8/300 code that Bound-T can or cannot analyse and so help you write analysable programs.

Some information in Chapters 3 and 6 of this Application Note applies only when the target-program executable is generated with specific compilers. These chapters discuss the GNU 'C' cross-compiler [6] and the IAR Systems 'C' cross-compiler [7]. Other compilers may be addressed in separate Application Notes.

This Application Note is applicable to various physical implementations (also known as devices, derivates, models or chips) of the H8/300 core architecture. However, the memory lay-out and timing parameters of the chip must be known to Bound-T.

1.2 Overview

The reader is assumed to be familiar with the general principles and usage of Bound-T, as described in the Bound-T User Manual . The user manual also contains a glossary of terms, many of which will be used in this Application Note.

So what's it all about?

In a nutshell, here is how Bound-T bounds the worst-case execution time (WCET) of a subprogram: Starting from the executable, binary form of the program, Bound-T decodes the machine instructions, constructs the control-flow graph, identifies loops, and (partially) interprets the arithmetic operations to find the "loop-counter" variables that control the loops, such as n in "for ($n = 1$; $n < 20$; n++) { ... }".

By comparing the initial value, step and limit value of the loop-counter variables, Bound-T computes an upper bound on the number of times each loop is repeated. Combining the loop-repetition bounds with the execution times of the subprogram's instructions gives an upper bound on the worst-case execution time of the whole subprogram. If the subprogram calls other subprograms, Bound-T constructs the call-graph and bounds the worst-case execution time of the called subprograms in the same way.

When does it work?

This sort of "static program analysis" is in theory an unsolvable problem and cannot work for *all* programs (like the well-known "halting problem"). It succeeds for programs that have a suitable structure, for example programs in which all loops have counters with constant initial and final values and a constant step. Moreover, since we are analysing low-level machine coder rather than high-level source code, the nature of the instruction set and the specific instructions used (or, usually, generated by the compiler) may help or hinder the analysis.

Stack usage analysis

In a similar way, Bound-T can analyse the machine code to find out where the stack pointer is changed and how much it is changed. From these changes, Bound-T can compute an upper bound on the stack usage of each subprogram, also including the stack-space used by called subprograms.

What follows

This Application Note explains how to use Bound-T to analyse H8/300 programs and how Bound-T models the architecture of this processor. To make full use of this information, the reader should be familiar with the architecture and instruction set of this processor, as presented in reference [4].

The remainder of this Application Note is divided into a *user guide* part and *reference* part.The user guide part consists of chapters 2 through 3 and is structured as follows:

- Chapter 2 shows how to use the H8/300 version of Bound-T. It briefly lists the supported H8/300 features and cross-compilers and fully explains those Bound-T command arguments and options that are wholly specific to the H8/300, or that have a specific interpretation for this processor.
- Chapter 3 addresses the user-defined assertions on target program behaviour and explains the possibilities and limitations in the context of the H8/300 and its cross-compilers.

The remainder of the Application Note forms the reference part as follows:

- Chapter 4 describes the main features of the H8/300 architecture and how they relate to the functions of Bound-T.
- Chapter 5 defines in detail the set of H8/300 instructions and registers that is supported by Bound-T.
- Chapter 6 explains which procedure calling protocols (ABIs) are supported by Bound-T.
- Chapter 7 listst all H8/300-specific warnings and error messages that Bound-T can issue and explains the possible reasons and remedies for each.

The implementation of Bound-T for the H8/300 is described in Samuel Peterssons's Master's thesis [9] and a related technical report [10].

1.3 References

[1] Bound-T User Guide.
Tidorum Ltd., Doc.ref. TR-UG-001.
http://www.bound-t.com/manuals/user-guide.pdf

[2] Bound-T Reference Manual.
Tidorum Ltd., Doc.ref. TR-RM-001.
http://www.bound-t.com/manuals/ref-manual.pdf

[3] Bound-T Assertion Language.
Tidorum Ltd., Doc.ref. TR-UM-003.
http://www.bound-t.com/manuals/assertion-lang.pdf

[4] H8/300 Programming Manual.
Renesas Technology Corporation, *http://www.renesas.com/*.
Originally published by Hitachi Ltd.
First edition, December 1989.

[5] H8/3297 Series Hardware Manual.
Renesas Technology Corporation, *http://www.renesas.com/*.

Originally published by Hitachi Ltd.
3rd edition, September 1997.

[6] GCC, the GNU Compiler Collection.
http://gcc.gnu.org/.

[7] H8/300 IAR C Compiler Reference Guide.
IAR Systems part number CH8300-1.
First edition, February 2000.

[8] H8/300 Application Binary Interface for GCC.
http://gcc.gnu.org/projects/h8300-abi.html.

[9] Porting the Bound-T WCET tool to Lego Mindstorms and the Asterix RTOS.
Master's thesis by Samuel Petersson, Mälardalen University, Västerås, Sweden, May 2005. *http://www.mdh.se/*.

[10] Using a WCET Analysis Tool in Real-Time Systems Education.
By Samuel Pettersson, Andreas Ermedahl, Anders Pettersson, Daniel Sundmark and Niklas Holsti. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-178/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University.

[11] Using Bound-T in HRT Mode.
Tidorum Ltd., Doc. ref. TR-UM-002.
http://www.bound-t.com/manuals/hrt-manual.pdf

1.4 Abbreviations and acronyms

See also references [1] and [2]for abbreviations specific to Bound-T and reference [4] for the mnemonic operation codes and register names of the H8/300.

ABI	Application Binary Interface, similar to a calling protocol.
C	Carry flag (in the CCR).
CCR	Condition Code Register.
PC	Program Counter, a dedicated register in the H8/300.
RAM	Random-Access Memory. Allows both reading and writing.
RAME	RAM Enable, the bit in the SYSCR that enables or disabled on-chip RAM.
Rn	One of the eight 16-bit general registers, n = 0 .. 7.
RnL	The low half (low octet) of register Rn, An 8-bit register.
RnH	The high half (high octet) of register Rn. An 8-bit register.
ROM	Read-Only Memory.
SP	The Stack Pointer, another name for R7.
SYSCR	System Control Register, a feature of some H8/300 chips.
TBA	To Be Added
TBC	To Be Confirmed
TBD	To Be Determined
V	Overflow flag (in the CCR).
WCET	Worst-Case Execution Time.
Z	Zero flag (in the CCR).

1.5 Typographic conventions

We use the following fonts and styles to show the role of pieces of the text:

Register	The name of a H8/300 register embedded in prose.
INSTRUCTION	An H8/300 instruction.
-option	A command-line option for Bound-T.
symbol	A mathematical symbol or variable.
`text`	Text quoted from a text / source file or command.

2 USING BOUND-T FOR H8/300

2.1 Input formats

Executable file

The target program executable file must be supplied in the standard GCC *COFF* format, or in the IAR proprietary *UBROF* format, or in the standard *S-record* format. Bound-T can usually determine the actual file type automatically, by inspecting the file, but the input file type can also be specified explicitly by means of the command-line options *-coff*, *-srec*, or *-ubrof*.

For COFF files, the "magic number" in the file header is expected to be hexadecimal 8300.

For S-record files, the supported format is defined below in section 2.4. S-record files carry no symbolic debugging information. To compensate, the addresses of subprograms and variables can be defined in a separate symbol definition file, introduced with the generic Bound-T option *-symbols* [2], or with the H8/300-specific option *-sym* which (for historical reasons) uses a slightly different symbol-file format defined in section 2.5 below.

For UBROF files, the last supported version known to work is UBROF 10. Versions before UBROF 7 may not work well.

Additional code and data file

The command-line option *-srec=file* can extend the memory image, as defined by the main target program file, with more code or data from an S-record file. This can be useful if the running program consists of a fixed ROM part (given in S-record form) and a variable application program (given as COFF, S-record, or UBROF).

Patch file

Sometimes it is useful to modify or “patch” the target program before analysis. Bound-T provides the generic option *-patch filename* that names a file that contains patches to be applied to the loaded target-program memory image before analysis starts. The format of the patch file is specific to the target processor. The H8/300 version of Bound-T does not support patching and thus no patch-file format is defined for patching H8/300 programs.

2.2 Command arguments and options

The generic Bound-T command format, options and arguments apply without modification to the H8/300 version of Bound-T. Please see the Reference Manual [2] for these.

Bound-T for the H8/300 is usually installed with the name *boundt_h8_300*, so a typical command will have the form

```
boundt_h8_300 <options> <program exe file> <subprogram names>
```

There are additional H8/300-specific options as explained in the table below, in alphabetical order. Note that a target-specific option must be written as one string with no embedded blanks, so the option-name and its numeric or string parameter, if any, are contiguous and separated only by the equal sign (=) but not by white space. For example, the form *-stack=internal* is correct, but *-stack = internal* is not.

Note that you must always specify the option *-device*, to choose the H8/300 device (also called the chip, model or derivative) for which the target program is meant. There is no default device.

Some H8/300 devices require or allow further device-specific options which are listed in a separate table in section 2.3.

Table 1: Command options

Option		***Meaning and default value***
-bcc=signed -bcc=unsigned	*Function*	Controls the interpretation of the condition codes for signed arithmetic in conditional branch (**Bcc**) instructions. Under *-bcc=signed,* the signed conditions are considered opaque (unknown to the analysis). This usually means that a loop using a counter of signed type cannot be bounded automatically. Under *-bcc=unsigned*, the signed conditions are interpreted as the corresponding unsigned conditions. This may allow automatic bounding of some loops using signed counters, but is safe only if the counter stays in the non-negative signed range. See section 5.5.
	Default	*-bcc=unsigned.*
-coff	*Function*	Tells Bound-T that the given target program file is a COFF file.
	Default	Bound-T tries to determine the file type by inspecting the file.
-device=*X* -device *X* -*X*	*Function*	Chooses the specific H8/300 device (model, chip) *X* on which the target program runs. This defines the memory lay-out and perhaps other processor properties. See the table in section 2.3 for a list of the supported devices and their names (*X*). The prefix '*-device*' or "*-device=*" is optional; you can also say just *-X*, for example *-H8/3292* or just *-3292*.
	Default	There is no default. **This option must be specified**.
-mint16 -mint32	*Function*	Tells Bound-T the size (16 bits or 32 bits) to assume for "int" variables in COFF programs compiled with GCC. This corresponds to the GCC options *-mint16* and *-mint32*.
	Default	*-mint16*
-stack=internal -stack=external	*Function*	Indicates that the stack (**SP** = **R7**) is placed in internal (on-chip) memory, giving fast access, or in external (off-chip) memory, giving slow access. Some devices support the option *-nternal_ram=disabled* which overrides *-stack=internal* and forces *-stack=external*.
	Default	*-stack=internal*
-read_ws=*X*	*Function*	Sets the number *X* of memory wait states assumed for an external memory read.
	Default	*-read_ws=0*
-srec	*Function*	Tells Bound-T that the given target program file is an S-record file following the syntax defined in section 2.4.
	Default	Bound-T tries to determine the file type by inspecting the file.
-srec=*file*	*Function*	Loads more code or data in S-record form from the named *file*, adding to the code and data read from the main executable file (as named on the command line after the options). The format of the file is described below in section 2.4. This option can be used at most once on the command line.
	Default	No additional S-record input.

Option		Meaning and default value
-sym=*file*	*Function*	Loads more symbol definitions from the named *file*, adding to the symbol tables read from the main executable file (as named on the command line after the options). For example, the file can define names for subprograms loaded form an S-record file (see -*srec*). The format of the *file* is described below in section 2.5. This option can be used at most once on the command line.
	Default	No additional symbol input.
-ubrof	*Function*	Tells Bound-T that the given target program file is an UBROF file from an IAR Systems compiler.
	Default	Bound-T tries to determine the file type by inspecting the file.
-ur*N*	*Function*	Here *N* is a digit from 0 to 3. This option tells Bound-T which *-ur* option was used when the target program was compiled with the IAR C compiler. The option changes the calling protocol with respect to which registers are used for parameters; see [7].
	Default	*-ur0* as in the IAR C compiler.
-uu*N*	*Function*	Here *N* is a digit from 0 to 4. This option tells Bound-T which *-uu* option was used when the target program was compiled with the IAR C compiler. The option changes the calling protocol with respect to callee-saved registers; see [7].
	Default	*-uu0* as in the IAR C compiler.
-write_ws=*X*	*Function*	Sets the number *X* of memory wait states assumed for an external memory write.
	Default	*-write_ws=0*

Table 2: H8/300-specific *-trace* items

-trace item	Traced information
load	Program elements (segments, sections, symbols, ...) as they are loaded from the executable file. May help to understand loading problems. If this option is selected when no root subprograms are named on the command line, the contents of the executable file are displayed twice: once while reading them, and once after the whole file has been read, as usual when no root subprograms are given.

2.3 Supported H8/300 devices

Why the specific device is important

There are several different models (chips, devices) of the H8/300. The choice of device is important for WCET analysis because the different devices have different memory lay-outs and so an access to the same address may use different kinds of memory and take different amounts of time on different devices. You must use the option *-device=X* to choose the device

(*X*) for which the target program is intended. For example, the option *-device=3292* chooses the H8/3292 device. Equivalent forms are *-device=lego*, or *-device lego*, or even just *-lego* or *-3292*.

Currently supported devices

The table below lists the devices that Bound-T currently knows about (supports).

Table 3: Supported H8/300 devices

Device	*Names for -device (case is not significant)*	*Ref.*	*Remarks*
H8/3292	H8/3292, 3292, Lego	[5]	The processor used in Lego Mindstorms™.
H8/3294	H8/3294, 3294	[5]	
H8/3296	H8/3296, 3296	[5]	
H8/3297	H8/3297, 3297	[5]	

Device-specific command options

Some H8/300 devices require or allow further device-specific options which are listed in the following table. To find out which options apply to a specific device *X*, run Bound-T with the options *-device=X -help*.

Table 4: Device-specific command options

Option		*Meaning and default value*
-internal_ram=enabled -internal_ram=disabled	*Function*	In some devices the program can disable the internal, on-chip RAM memory by clearing the RAME bit in the System Control (SYSCR) register. In this state the internal RAM addresses instead access external memory which is slower. This option tells the analysis what to assume about RAME. Note that *-internal_ram=disabled* implies *-stack=external*.
	Default	*-internal_ram=enabled.*
-mode=1 -mode=2 -mode=3	*Function*	Defines the processor's operating mode, which influences the memory map. For the H8/3297 series chips the mode is set by some input pins, see ref. [5]. Mode 1: expanded mode without on-chip ROM. Mode 2: expanded mode with on-chip ROM. Mode 3: single-chip mode (no external memory). Note that mode 3 implies *-stack=internal*.
	Default	*-mode=2*.

2.4 S-Record file format

A target program in S-record format

Bound-T for the H8/300 can read target programs in an *S-record* format, a common textual file format for memory images. Bound-T can generally detect the format of the target program automatically; if that does not work, the command-line option *-srec* can be used to make Bound-T assume S-record format for the target program file.

An additional code or data file in S-record format

A (main) target program file (in any format) can be augmented by one S-record file, named with the command-line option *-srec=file*. This option makes Bound-T read additional target memory contents (code and/or data) from the named *file* which is assumed to contain S-records. Bound-T creates the memory image of the target program (as it would be at load time, before target execution is started) by combining the memory contents from the main executable file (named on the command line after the options) and this S-record file. If the memory areas defined by the two files overlap, Bound-T emits an error message because it cannot know which file, if either, should be given priority.

The S-record format

An S-record file should be a text file where each line is an S-record. An S-record is here defined as a character string that contains five fields as follows:

- *Type* field, two characters, "S1" .. "S9".
- *Record length*, two hexadecimal digits, giving the number of two-digit hexadecimal numbers that follow in this record (excluding the *Type* and *Record length* fields). In other words, the number of characters remaining in this record is twice this number.
- *Address*, four hexadecimal digits, giving the load address of the first data octet.
- *Data*, a number of code or data octets, each encoded as a two-digit hexadecimal number. The number of data octets is (*Record length*) – 3, where the subtracted constant 3 represents two octets of *Address* and one octet of *Checksum*.
- *Checksum*, two hexadecimal digits defining the least significant octet of the one's complement of the sum of all the octets in the *Record length*, *Address* and *Data* fields.

The following is an example of an S-record:

```
    S11301506DF60D566D750D554B140D664A201786B8
```

In this example, the *Type* field is "S1", the *Record length* field is "13" which means 19 (decimal) more two-digit hexadecimal numbers, the *Address* field is "0150" which means the address 336 (decimal), the *Data* field contains the sixteen two-digit hexadecimal numbers "6DF6 ... 1786", and the *Checksum* field is the last two characters, "B8".

The meaning of the different record types may depend somewhat on the software that generates and uses the files. However, generally "S1" lines are data/code records as described above. In a typical S-record file, all lines are "S1" lines except perhaps for the last line, which may be an "S9" termination record. In an "S9" record the *Address* field may contain the start/entry address for the code given in the "S1" records and there is no *Data* field.

Bound-T handles all S-records in the same way, by loading the *Data* field into the target program's memory image starting at the *Address* given in the S-record.

2.5 H8/300-specific symbol-file format

The generic -symbols option

Bound-T has a generic option *-symbols* for naming additional symbol-definition files to complement the set of target-program symbols defined in the executable target program file. This generic option works also in Bound-T for the H8/300. The option and the generic format of symbol-definition files are described in the Bound-T Reference Manual [2].

The H8/300-specific -sym option

For historical reasons and upwards compatibility there is a similar H8/300-specific option that is written *-sym=file*. This option makes Bound-T read additional symbol definitions from the named *file*. This *file* uses an H8/300-specific format, which is similar to, but not exactly the same as, the format for the generic *-symbols* option. All symbols in a *-sym* file represent subprograms; symbols for data variables cannot be defined with this format.

The file named in the *-sym* option should be a text file with one symbol definition per line. Leading and trailing blanks are ignored. Blank or empty lines are ignored as is any line beginning with two hyphens (--) possibly preceded by blanks. Horizontal tabulation characters (HT, TAB) are equivalent to blanks.

A symbol definition associates a subprogram identifier (the symbol) with a memory address and thus contains two strings, without embedded blanks but separated by one or more blanks, as follows:

- The first string is the identifier, possibly qualified by scope, using the default scope delimiter.
- The second string is the numerical memory address of the symbol expressed in Ada literal form. It can thus be in decimal form, for example 1234, or in based form, for example in hexadecimal as 16#ab26#.

The strings in a symbol definition should not be enclosed in quotes (unlike symbols in assertion files). The following is an example of a symbol file:

```
-- This is a comment.
rom|init_mem     16#1a7#
rom|boot|startup 0
```

The first line is a comment and Bound-T ignores it. The second line defines the subprogram symbol *init_mem*, in the scope (module) *rom*, and gives it the address 1a7 hexadecimal = 423 decimal. The third line defines the subprogram symbol *startup*, in the scope *rom* and the sub-scope *boot*, and gives it the address (decimal) zero.

2.6 HRT analysis

Regarding HRT analysis there are no specific considerations for the H8/300. Please refer to the Bound-T Reference Manual [2] and the Bound-T HRT analysis manual [11].

2.7 Choice of procedure calling protocol

The analysis of the computations in a subprogram, and in the subprograms that call this subprogram, depend on the *calling protocol* of the subprogram.

Bound-T chooses the calling protocol as follows, depending on the format of the target program executable file being analysed:

- If the executable file is in COFF or S-record form the GCC calling protocol is used for all subprograms.
- If the executable file is in UBROF form the IAR calling protocol is used for all subprograms.

If necessary, Tidorum will extend Bound-T/H8/300 so that the calling protoocol of a subprogram can be defined by an assertion on a specific property for this subprogram. See property assertions in [3].

2.8 Meaning of results

Execution time (WCET)

The WCET is given in units of execution states = clock periods.

The WCET reported for an H8/300 subprogram corresponds to an execution from the first instruction at the subprogram's entry point, up to and including the last instruction (**RTS** or **RTE**) of the subprogram. It does not include the caller's side of the calling sequence (pushing parameters on the stack or loading them into registers) nor the post-call sequence in the caller (popping parameters off the stack).

Stack usage

Bound-T for the H8/300 analyses only the usage of the H8/300 native stack, for which the register **SP** = **R7** is the stack pointer. Possible compiler-defined or application-defined "soft-ware" stacks are not analysed.

The stack usage is given in octet units, although one should push and pop words [4]. Thus, the stack usage is normally an even number of octets.

The stack usage reported for a subprogram does not includes the pushed return address (pushed **PC**). The pushed return address is counted in the stack usage of the *calling* subprogram (the one that executes the **JSR** or **BSR**).

When an interrupt (exception) happens the processor pushes the **CCR** and the **PC** on the stack and then enters the interrupt handler. This pushes 4 octets on the stack (a padding octet is pushed together with the **CCR**). The total stack usage for an interrupt is thus 4 octets more than Bound-T reports reports as the stack usage for the interrupt-handler subprogram.

3 WRITING ASSERTIONS

3.1 Overview

If you use Bound-T to analyse non-trivial programs you nearly always have to write *assertions* to control and guide the analysis. The most common role of assertions is to set bounds on some aspects of the behaviour of the target program, for example bounds on loop iterations, that Bound-T cannot deduce automatically. Assertions must identify the relevant parts of the target program, for example subprograms and variables. The Bound-T assertion language has a generic high-level syntax [3] in which some elements with target-specific syntax appear as the contents of quoted strings:

- subprogram names,
- code addresses and address offsets,
- variable names,
- data addresses and register names,
- stack names,
- instruction roles, and
- names of target-specific properties of program parts.

In practice the *names* (identifiers) of subprograms and variables are either identical to the names used in the source code, or some “mangled” form of the source-code identifiers where the mangling depends on the cross-compiler and not on Bound-T. However, Bound-T defines a target-specific way to write the *addresses* of code and data in assertions. *Register names* are considered a kind of “data address” and are target-specific.

This chapter continues the user-guide part of this Application Note by defining the H8/300-specific aspects of the assertion language. This chapter explains any specific limitations and possibilities for user-specified assertions when Bound-T is used with H8/300 programs.

3.2 Symbolic names

Linkage symbols

When the target program is compiled with debugging, the executable file usually contains a symbol-table that Bound-T can use to connect the symbolic names of subprograms and variables to their machine-level addresses for the analysis. You can then write assertions using the symbolic names. (Executable files in S-record form are an exception and do not contain any debugging information. The options *-symbols* or *-sym* can be used to define symbolic names even for S-record files.)

As in most versions of Bound-T, you must use the *linkage symbols*, not the source-code identifiers, to name subprograms and variables. Depending on the compiler and linker, the linkage symbols may be the same as the source-code identifiers or they may have some additional decoration or mangling. For example, some C compilers add an underscore at the front of the source-code identifier, so a C function called *foo* will have the linkage symbol *_foo*. The assertions must use the latter form, for example

```
subprogram "_foo" ... end "_foo";
```

To find out the linkage names in the target program, you can either dump the executable file (run Bound-T with just the executable file name as argument) or ask Bound-T to list all the subprogram and variable names by running Bound-T in the normal way but with the option *-trace symbols*.

Scopes

Programs often contain many variables with the same name, in different lexical *scopes*, that is, in different subprograms, blocks, or file scopes. In the assertion language, the symbolic name of a subprogram or variable can be prefixed with a scope string to show which of the several entities with this name is meant.

The H8/300 version of Bound-T uses the normal lexical scopes of symbolic identifiers, which are source-file (or module) name, subprogram name, and block name. Details may depend on the compiler and executable file format.

For example, if the C functions *foo* and *bar* both contain a variable *num*, you would write, in an assertion, `"foo|num"` for the first, and `"bar|num"` for the second.

3.3 Naming items by address

Subprograms, labels, exception vectors

Subprograms and labels can be named (identified) by the hexadecimal address, in quotes, without any prefixes like “0x” or the like. For example, if subprogram *foo* is located at 12AC hex (that is, this is the entry address of *foo*) then *foo* can be identified by `"12AC"` or `"12ac"`.

To identify a subprogram or label in an assertion, the identifying address should be preceded by the "address" keyword. You can also identify a root subprogram on the command line by its address, but do not use the "address" keyword in this case.

For example, this assertion bounds a loop in subprogram *foo*, assuming that *foo* starts at 12AC:

```
subprogram address "12AC"
  loop repeats 91 times; end loop;
end "12AC";
```

To analyse the subprogram that starts at 12AC, in the target program file *prog.coff*, use this Bound-T command; note the absence of an "address" keyword:

```
boundt_h8_300 -device=3297 prog.coff 12AC
```

Depending on the command shell that you use, it may or may not be allowed to put quotes around the address on the command line.

Code-address offsets

Some forms of assertions define code addresses by giving a *code offset* relative to a base address. For Bound-T/H8/300 a code offset is written as a hexadecimal number possibly preceded by a sign, '–' or '+', to indicate a negative or positive offset. If there is no sign the offset is considered positive.

Assume, for example, that the subprogram *Rerun* has the entry address 14AC hexadecimal and the subprogram *Abandon* has the entry address 15A0 hexadecimal. The subprogram with the entry address 14D4 hexadecimal can then be identified in any of the following ways, among many others:

- Using the absolute address:

```
subprogram address "14D4"
```

- Using a positive hexadecimal offset relative to the entry point of *Rerun*:

```
subprogram "Rerun" offset "28"
```

- Using a negative hexadecimal offset relative to the entry point of *Abandon*:

```
subprogram "Abandon" offset "-CC"
```

Note that the sign, if used, is placed within the string quotes, not before the string.

Variables, registers, memory locations

Assertions can name H8/300 storage cells directly, without using a source-level symbolic identifier. This is done by using the "address" keyword, followed by a quoted string that defines the storage cell. The syntax is described in the table below. The syntax is not case-sensitive, so "r4" is the same as "R4".

Table 5: Naming storage cells

Storage cell	*Syntax (without quotes)*	*Example*	*Meaning*
Word register	`R`<*number 0 .. 7*>	`R0`	Word register **R0**.
Octet register	`R`<*number 0 .. 7* ><`L` *or* `H`>	`R4L`	Low octet of **R4**.
Octet in memory	`B`<*hex address*>	`B34a`	Octet at memory address 34A hex.
Word in memory	`W`<*hex address*>	`W12C4`	Word at memory address 12C4 hex.
Octet parameter	`PB`<*decimal offset*>	`PB3`	Octet parameter in the stack, at an address 3 octets higher than the address of the stacked return address.
Word parameter	`PW`<*decimal offset*>	`PW2`	Word parameter in the stack, at an address 2 octets higher than the address of the stacked return address.
Octet local variable	`LB`<*decimal offset*>	`LB1`	Local octet in the stack, at at an address 1 (one) octets less than the address of the stacked return address.
Word local variable	`LW`<*decimal offset*>	`LW2`	Local word in the stack, at an address 2 (two) octets less than the address of the stacked return address.

Some examples of assertions on storage cells:

```
variable address "R3" 0 .. 100; -- Register R3 bounded.
variable address "r3" 0 .. 100; -- Same thing.


variable address "b3fa7" 20;
-- The octet at the nemory address hex 3FA7 (= decimal 16295)
-- has the value 20 (decimal).
```

```
variable address "pw2" 55;

-- The parameter word that was pushed immediately before
-- the JSR has the value 55
```

3.4 Stacks

Currently Bound-T for H8/300 supports only the standard H8/300 stack, pointed to by the **SP** register (**R7**). This stack is called “SP”, which is the stack-name to be used in assertions on stack usage and final stack height. However, since only one stack is defined, stack-usage assertions can also omit the stack-name.

3.5 Instruction roles

The generic assertion language [3] contains syntax for asserting the "role" that a given instruction (identified by its address or offset) performs in the computation, for example whether an instruction performs a branch or a call. The roles and their names are target-specific. The H8/300 version of Bound-T defines no assertable roles, thus such assertions cannot be used.

3.6 Assertable properties

The generic assertion language [3] contains syntax for asserting the value of target-specific "properties" for various parts of the target program. The assertable properties for the H8/300 are listed and explained in the following table.

Table 6: Assertable properties

Property name		*Meaning, values and default value*
`bcc_signed`	*Function*	Can be asserted for a subprogram, to define the model used for signed **Bcc** conditions within this subprogram, possibly overriding the command-line option *-bcc*.
	Value	The value 0 means to use whatever value of *-bcc* is set on the command line, or the default value of *-bcc* if there is no *-bcc* option on the command line. The value 1 means to model signed conditions as unknown values (as under *-bcc=signed*).
	Default	The behaviour set by the command-line option *-bcc*.

4 THE H8/300 PROCESSOR AND TIMING ANALYSIS

This chapter starts by describing the H8/300 processor. The focus is on how the H8/300 architecture and instruction set are likely to be used for the coding of loops, loop counters and accesses to subprogram parameters, which are the most important aspects for analysing the worst-case execution path and worst-case stack usage.

Then we explain in a general way how Bound-T models and analyses this processor. Some registers and instructions are modelled exactly, others approximately, and some not at all (or very approximately). Two aspects of the model and analysis are important:

- The effect of each instruction on the computation and on the control flow.
- The contribution of each instruction to the total execution time.

Chapter 5 then explains the supported features and the model and analysis in more detail.

4.1 H8/300 architecture and instruction set

The H8/300 [4] is an 8/16-bit microcontroller. It has a "von Neumann" architecture (a common memory for program and data). The memory address is 16 bits so at most 64 kilo-octets can be addressed (without paging schemes).

General registers

The processor has eight 16-bit general registers **R0** .. **R7**. The registers can also be accessed as sixteen 8-bit registers by appending **L** or **H** to the register name to specify the low or high octet. Register **R7** has the special function of a stack pointer and is also known as **SP**.

Instruction set

The instruction set provides 8-bit and 16-bit integer addition, subtraction, multiplication (8 bits times 8 bits giving a 16-bit product) and division (16 bits divided by 8 bits giving an 8-bit quotient and an 8-bit remainder).

The instruction set is complete enough that one can use 8-bit-wide instructions (eg. **ADD.B**) for most 8-bit computations and 16-bit-wide instructions (eg. **ADD.W**) for most 16-bit computations.

However, the instruction set is not fully orthogonal for data width; some kinds of instructions allow only 8-bit data and others only 16-bit data. For example, there addition, subtraction or comparison instructions that work on a 16-bit immediate value and a 16-bit register. This means that some 16-bit computations must be built up from 8-bit-wide instructions. For example, to add a 16-bit immediate value to a 16-bit register one would use the **ADD.B** instruction to add the low octet of the immediate to the low octet of the register, followed by an **ADDX** (add with carry) instruction to add the high octet of the immediate value to the high octet of the register, including the possible carry from the **ADD.B**.

Signed integers are represented in two's complement notation as usual, so the same **ADD** and **SUB** instructions apply to both signed and unsigned data. The conditional branch instruction **Bcc** provides different condition codes for signed and unsigned comparisons. Also the multiplication and division instructions are sensitive to signedness.

There are also instructions for processing individual bits. The core H8/300 does not support hardware floating point operations.

Program Counter and Condition Code Register

The 16-bit Program Counter (**PC**) register points to the next instruction. Instructions are 16 or 32 bits in length and always start at an even address.

The 8-bit Condition Code Register (**CCR**) contains the usual condition flags: **Z** = zero, **C** = carry, **N** = negative, **V** = overflow, **H** = half-carry (for BCD arithmetic). The **CCR** also contains the interrupt mask bit and two user-defined flag bits.

Stack Pointer

The Stack Pointer **SP** (**R7**) has all the functionality of a general register but is implicitly used by the call and return instructions and interrupt handling. Parameters can be passed on the stack and local variables can be held on the stack. The stack can be placed anywhere in the memory space (except in ROM, of course).

The stack grows down in memory (a push decreases the **SP**). The stack is a "full" stack which means that **SP** points to the topmost occupied stack location (as opposed to the first empty location).

Typically, the caller pushes parameters on the stack before the call; the call instruction then pushes the return address; and the callee pushes its local variables. Local variables and parameters can then be accessed with a positive offset from **SP** (addressing mode register-indirect with displacement).

The push and pop instructions operate on word (16-bit) data. The H8/300 manual [4] advises that the stack pointer should always be altered in word units and thus **SP** should always have an even value.

Memory addressing modes

Memory is addressed by octet but single instructions can access octets or 16-bit words. Words are stored in big-endian form; the high octet is always at an even address and the low octet at the next odd address.

The addressing modes (forms of instruction operands) include immediate, absolute address, register and register indirect. The register-indirect mode can include a constant displacement or auto-increment or auto-decrement. Branches and calls also support **PC**-relative and memory indirect (vectored) operands.

The H8/300 architecture is not (visibly) pipelined. This means that jumps and branches take effect immediately; there are no "delay slots" and jump or branch instructions.

4.2 Instruction timing in the H8/300

Execution time in H8/300 is counted in *execution states;* one execution state corresponds to one *clock period*.

The execution time of an instruction depends on the type of the instruction and on the kind of memory that the instruction accesses. The fastest instructions take two states; for example an **ADD.B** or **ADD.W** executed from the on-chip memory. Complex instructions that access memory may have execution times of 20 or more states, depending on the length of the instruction, the addressing mode, the data width and the memory areas that are accessed.

Instruction timing is usually independent of the processed data (register or memory values). The exception is the instruction **EEPMOV** that copies a block of data from one memory address to another; here the execution time depends on the size of the block, which is given in a register.

Memory areas

The H8/300 has three kinds of memory area:

- *On-Chip Memory*. Most of this is usually ROM with a smaller part of RAM. An access takes 2 execution states, whether for read or write and for either octet or word data.
- *On-Chip Register Field*. This area contains memory-mapped peripheral registers. An octet access takes 3 states and a word access (including instruction fetches and stack accesses) takes 6 states.
- *External Memory*. This is ROM or RAM that is not on the same chip as the processor. Access time is the same as for the On-Chip Register Field plus possible wait states.

The address range for the several areas — the memory map — depends on the model of the processor, that is, the specific chip, and perhaps also on the "operating mode" and status settings of the chip. The on-chip memory is usually at the start of the address range and the on-chip register field is at the end. You must use the command-line option *-device=X* to tell Bound-T which device is used.

When the device is known, for an access to a statically known memory address Bound-T can look up the kind of memory at this address and determine the correct access time. For a dynamic (register indirect) memory address Bound-T cannot always determine the memory area and must then assume the worst case (external memory or on-chip register field).

Disabled on-chip RAM

In some H8/300 devices the program can disable or enable the internal (on-chip) RAM memory by clearing or setting the **RAME** bit in the System Control Register (**SYSCR**). When the internal RAM is disabled, addresses normally mapped to the internal RAM will instead access external memory and so are slower.

Bound-T assumes that the **RAME** setting is constant for all code in one analysis, according to the command-line option *-internal_ram=enabled* or *-internal_ram=disabled*. This lets Bound-T know whether to assume fast (internal) or slow (external) access time for "internal RAM" addresses. If this assumption is false, and the program instead switches back and forth between enabled and disabled internal RAM, you must specify *-internal_ram=disabled* to be safe with respect to the access time in the disabled state, but this will over-estimate the access time in the enabled state. Note, moreover, that dynamic changes in the enabled or disabled state of the on-chip RAM is equivalent to memory bank switching and may invalidate even the control-flow analysis of Bound-T.

Input-output instructions

There are two special instructions, **MOVFPE** and **MOVTPE**, that synchronize with the peripheral E (enable) clock and therefore have a variable execution time. The variable component is the operand access time which varies from 9 to 16 execution states.

Summary

To summarize, the following H8/300 architectural features can lead to approximate (over-estimated) execution times for the concerned instructions:

- Memory access time for dynamic addressing when Bound-T cannot determine the area that is accessed.
- Run-time changes to the enabled or disabled state of the internal (on-chip) RAM memory.
- The instructions **MOVFPE** and **MOVTPE**.

One instruction, **SLEEP**, can lead to an underestimated execution time because the duration of the "sleep mode" is not included in the WCET bound. Bound-T warns about this when it find a **SLEEP** instruction in the code under analysis.

Finally, do not forget that even if the timing of each instruction is exactly modelled, over-estimates can also result from general features of the program under analysis, for example long execution paths that are logically infeasible, but which Bound-T includes in the worst-case time because it does not detect the logical infeasibility.

5 SUPPORTED H8/300 FEATURES

5.1 Overview

This chapter explains in detail how Bound-T models H8/300 instructions, registers and status flags. We will first describe the extent of support in general terms, with exceptions listed later. Note that in addition to the specific limitations for the H8/300, Bound-T also has generic limitations as described in the Bound-T User Guide [1] and Reference Manual [2]. For reference, these are briefly listed in section 5.1.

General support level

In general, when Bound-T is analysing a target program for the H8/300, it can decode and correctly time all instructions, with minor approximations except for the effects of memory access time.

Bound-T can construct the control-flow graphs and call-graphs for all instructions, assuming that the program obeys one of the supported procedure calling protocosl listed in chapter 6 and does not change the enabled/disabled state of the on-chip RAM memory while executing code from this part of the address space. Note that there are generic limitations on the analysis of jumps and calls that use a dynamically computed target address or a dynamically computed return address.

When analysing loops to find the loop-counter variables, Bound-T is able to track all the computations that use *unsigned* integer values (8 or 16 bit) and additions and subtractions. However, sequences of 8-bit or 16-bit operations that use carry/borrow bits to build up arithmetic wider operands are not tracked in general. For example, a 16-bit addition could be built from an 8-bit **ADD.B** followed by an 8-bit-with-carry **ADDX**. Bound-T understands such a combination only in some special cases as explained in section 5.3.

Bound-T correctly detects when tracked integer computation is overridden by other computations in the same registers, such as multiplication, division, bit-wise boolean operations or single-bit operations. Note that there are generic limitations on the analysis of pointers to counter variables.

Signed integer arithmetic is not completely supported (because of the complexity of the mixed 8/16-bit register set). However, signed arithmetic should be correctly analysed if the loop-counter values stay in the non-negative range: 0 to 127 for 8-bit counters and 0 to 32 767 for 16-bit counters. See also the option *-bcc=signed* in section 2.2.

Although arithmetic is usually considered unsigned, Bound-T interprets immediate operands as signed two's complement numbers, to model decreasing loop counters.

In summary, for a program written in a compiled language such as Ada or C, with a compiler that uses one of the supported procedure calling standards, and under the restriction that loop counters are unsigned or use only the non-negative range of a signed intgers, it is unlikely that the Bound-T user will meet with any constraints or limitations that are specific to the H8/300 target system.

Before detailing the exceptions to the general support, some terminology needs to be defined concerning the levels of support.

Reminder of generic limitations

To help the reader understand which limitations are specific to the H8/300 architecture, the following compact list of the generic limitations of Bound-T is presented. See the User Manual for a full description of the generic limitations.

Table 7: Generic limitations of Bound-T

Generic Limitation	*Remarks for H8/300 target*
Understands only integer operations in loop-counter computations.	No implications specific to the H8/300. However, note that in the H8/300 the integers are further limited to unsigned types or to the non-negative range of signed types.
Understands only addition, subtraction and multiplication by constants, in loop-counter computations.	No implications specific to the H8/300.
Assumes that loop-counter computations never suffer overflow.	No implications specific to the H8/300. However, overflow is perhaps more likely to occur, on mistake or on purpose, in arithmetic with a small number of bits, such as the 8 or 16 bits in the H8/300.
Can bound only counter-based loops.	No implications specific to the H8/300.
May not resolve aliasing in dynamic memory addressing.	No implications specific to the H8/300.
May ascribe the wrong sign to an immediate (literal) constant operand.	No implications specific to the H8/300.

5.2 Support Synopsis

The following table gives a synoptical view of the supported H8/300 features. The features are ordered from the fully supported at the top, to the unsupported at the bottom. More detail on the support level is given in the following sections.

The table lists 8-bit and 16-bit instructions separately, but in fact these interact because the 8-bit registers are part of the 16-bit registers. This interaction is explained in section 5.3.

Table 8: Synopsis of H8/300 support

H8/300 instruction	*Remarks*
ADD.W, **SUB.W**, **ADDS**, **SUBS**, **CMP.W**, **MOV.W**, **POP**, **PUSH**	Unsigned 16-bit integer arithmetic is assumed. However, immediate operands are taken as signed 2's complement numbers. The option *-bcc=signed* enables analysis of signed conditional branches assuming that the integer values are in the non-negative signed range 0 .. 32 767.
ADD.B, **SUB.B**, **INC**, **DEC**, **CMP.B**, **MOV.B**	Unsigned 8-bit integer arithmetic is assumed. However, immediate operands are taken as signed 2's complement numbers. The option *-bcc=signed* enables analysis of signed conditional branches assuming that the integer values are in the non-negative signed range 0 .. 127.
ADD.B #i, RnL followed by **ADDX #j, RnH**	Modelled as the 16-bit operation **ADD.W #j:i,Rn**, an instruction that is not implemented in the H8/300.
SUB.B #i, RnL followed by **SUBX #j, RnH**	Modelled as the 16-bit operation **SUB.W #j:i,Rn**, an instruction that is not implemented in the H8/300.

H8/300 instruction	*Remarks*
CMP.B #i, RnL followed by **SUBX #j, RnH**	Modelled as the 16-bit operation **CMP.W #j:i,Rn**, an instruction that is not implemented in the H8/300. Also, RnH becomes opaque since the **SUBX** changes it.
XOR when both operands are the same (octet) register	Equivalent to assigning zero to the register.
INC, **DEC**	Modelled as integer increment and decrement. The **Z** flag is modelled correctly if there is at most one **INC** overflow (wrap-around from 255 to 0) and at most one **DEC** underflow (wrap-around from 0 to 255) in any loop that uses these instructions for counting iterations.
POP, PUSH	The accessed stack location depends on the local stack height, which may be unknown, in which case the analysis is incomplete.
JMP with static address	Modelled exactly.
JMP @Rn with static address table	See section 5.6.
JMP @@aa:8 with static vector **JSR @@aa:8** with static vector	See section 5.6.
Bcc	The modelling of the condition codes for signed comparisons depends on the option *-bcc*. See section 5.5.
BSR, **JSR** with static address	The arithmetic effect of the **BSR** or **JSR** instruction itself is modelled, but the effect of the called subprogram is modelled only on the definition level (which cells are changed but not which values they get). This is a general Bound-T feature, not specific to the H8/300.
RTS, **RTE**	Modelled exactly, assuming that the returning subprogram follows the adopted calling protocol. See chapter 6.
EEPMOV	The instruction is expanded into a loop in the flow-graph. The loop models the arithmetic effect on the registers (**R4H**, **R5**, **R6**) but not the effect on the destination memory because the dynamic memory access (**@R6**) is usually not resolved to known memory locations.
NOP, **SLEEP**	The time spent in sleep mode is not included in the reported WCET bound.
ANDC, **ORC**, **XORC** **LDC** with immediate operand	The effect of the immediate operand on the **CCR** flags **Z** and **C** is fully modelled. Other **CCR** flags are not modelled at all.
AND, **OR**, **XOR**	Modelled exactly. However, the arithmetic analysis in Bound-T has generic limitations on modelling bitwise logical operators when used in loop counting.
NEG	Result is opaque because we model unsigned arithmetic. However, the **Z** flag is set to "operand = 0" and the **C** flag is set to "operand > 0".
MULXU	Result is opaque, but can become non-opaque if constant propagation constrains one operand to a single static value.
ADDX, **SUBX**	Result is opaque because these instructions are used to build multi-octet addition and subtraction which depends on overflow (carry/borrow) in the addition and subtraction of individual octets. Note the special case above for **ADDX** immediately after **ADD.B** and **SUBX** immediately after **CMP.B**.

H8/300 instruction	*Remarks*
DAA, **DAS**, **DIVXU**	Result is opaque.
ROTL,**ROTR**, **ROTXL**, **ROTXR**, **SHAL**, **SHAR**, **SHLL**, **SHLR**	Result is opaque.
BAND, BIAND, BILD, BIST, BLD, BNOT, BOR, BSET, BTST, BXOR	Result is opaque.
LDC with register operand	The **Z** and **C** flags become opaque.
STC	The destination register becomes opaque.
Changing the enabled/disabled state of the on-chip RAM when accessing data in this part of the address space	The enabled/disabled state in effect chooses between two memory banks, on-chip and off-chip. Bound-T does not track this change and so its data-flow analysis is wrong.
JMP @Rn **JMP @@aa:8** with dynamic vector	If the dynamic target address is not resolved, the WCET bound for this subprogram omits the rest of the control flow (the **JMP** is taken as a return from the subprogram).
JSR @Rn **JSR @@aa:8** with dynamic vector	Modelled as a call with a dynamically computed target address.
Changing the enabled/disabled state of the on-chip RAM when executing code from this part of the address space	The enabled/disabled state in effect chooses between two memory banks, on-chip and off-chip. Bound-T does not track this change and so its understanding of which instructions are fetched and executed is wrong.
Self-modifying code	Not supported.

5.3 Registers and memory locations

This section explains how Bound-T models the H8/300 registers and memory. Chapter 6 describes the additional support when some specific procedure calling protocol is in use.

General registers R0 .. R7 and their low and high octet parts

All H8/300 general registers (**R0** .. **R7**, **R0L** .. **R7L**, **R0H** .. **R7H**) are supported fully by Bound-T as operands and destination registers for instructions. Each register is modelled as a separate data cell, but an automatic connection is defined between the word registers and their octet parts as follows:

- When an octet-wide instruction assigns a value to an octet register, Bound-T includes an implied assignment of an opaque value to the corresponding word register. For example, an instruction with destination **R4H** or **R4L** implies that R4 receives an opaque value.
- When a word-wide instruction other than **MOV.W** assigns a value to a word register, Bound-T includes implied assignments of opaque values to the octet parts of the word register. For example, an instruction with destination **R2** implies that **R2L** and **R2H** receive opaque values.
- When a **MOV.W** copies an immediate operand (a static literal value) or a 16-bit register or a memory word into another 16-bit register or memory word, Bound-T includes implied assignments of the corresponding octet parts. For example, the instruction **MOV.W R1, R6**, with the principal effect **R6** := **R1**, implies the assignments **R6L** := **R1L** and **R6H** := **R1H**.

Combining 8-bit operations into 16-bit operations

When a processor only supports short arithmetic, such as 8 or 16 bits on the H8/300, arithmetic on longer operands is built up by a sequence of short operations connected by carry/borrow propagation. Bound-T does not have a general mechanism for tracking such instruction sequences and modelling the effect on the long operands. Happily on the H8/300, 16-bit arithmetic can usually be implemented with dedicated 16-bit instructions. The exception is the lack of immediate 16-bit operands for addition, subtraction and comparison, which therefore must be implemented as a pair of 8-bit instructions. Such pairs are often applied to 16-bit loop counters and Bound-T therefore tries to detect them and to provide a 16-bit model of the arithmetic. The following table shows these instruction pairs and their model.

Table 9: Paired 8-bit Instructions

Instruction pair	*Arithmetic model*	*Remarks*
ADD.B #i, RnL **ADDX #j, RnH**	If the constant *j:i* is non-negative: **Rn** := **Rn** + constant *j:i* **Z**, **C** := unknown If the constant *j:i* is negative: **Rn** := **Rn** – abs (constant *j:i*) **Z** := result = 0 **C** := result < 0	The constant *j:i* is the 16-bit value composed of *j* in the high octet and *i* in the low octet. It is considered non-negative if *j* < 128 and negative otherwise (that is, if bit 15 is 1).
CMP.B #i, RnL **SUBX #j, RnH**	**Z** := **Rn** = constant *j:i* **C** := **Rn** < constant *j:i* **RnH** := unknown	The constant is always taken as unsigned.

Memory data

Memory locations with statically know address are fully supported as operands and destinations for instructions. Bound-T separates between memory octets and memory words: a different logical storage cell models the octet at a given address and the word at the same address. However, an assignment to a cell that models a memory word also implies an update of the two cells that model the octets in the word, and an assignment to a memory octet cell makes the containing word cell opaque.

For memory accesses with a dynamic address, for example in instructions like **MOV R3,@R5**, Bound-T uses arithmetic analysis to try to bound the address to a single value. If this succeeds, the arithmetic model is refined to use the statically addressed memory cell. (Dynamic accesses based on the Stack Pointer **SP** = **R7** are a special case as explained below.)

Unresolved dynamic memory write accesses are ignored at present. Thus, an instruction that writes to an unresolved dynamically computed address is assumed to have no effect on any the relevant computation, which is an unsafe approximation. An instruction that reads from an unresolved dynamic address yields an opaque value, which is a safe assumption.

A word operand in the H8/300 memory is always aligned on an even octet address. However, the program can use an odd address to access a word, because the processor ignores the least significant bit and acts as if the address were even. Bound-T takes this into account: all word-sized memory cells are identified by an even address, even if the code uses and odd (static) address.

Stack data

On the H8/300, a subprogram usually stores its local variables in the stack and accesses them using Register Indirect mode based on the Stack Pointer **SP** = **R7**. Some of the parameters to a subprogram can also be passed in the stack and accessed in this way.

The **SP** is usually initialized in the target program's boot/reset routine and then varies as subprograms are entered and exited and when parameters and local variables are pushed or popped. A Bound-T analysis is usually rooted at a subprogram and seldom starts from the boot/reset point. This means that the absolute value of **SP** is usually unknown during the analysis of a given subprogram; we only know how the **SP** changes within the subprogram. Dynamic memory accesses based on **SP** are therefore analysed in special way as follows.

The *local stack height* at a given point in a subprogram is defined as the difference between the value of **SP** at this point and the value of **SP** on entry to the subprogram, with the sign chosen so that a stack push increases the local stack height. This means that the return address pushed by a call instruction is counted in the stack usage of the caller, not that of the callee. When a subprogram returns, it usually pops the return address off the stack, resulting in a final local stack height of -2 octets.

Register-indirect memory references based on **SP** are considered statically resolved when the local stack height is known at the instruction that contains the reference. The accessed memory location is identified by its offset relative to the value of **SP** on entry to the subprogram; this offset is the difference between the local stack height and the **SP**-relative displacement encoded in the accessing instruction. Bound-T models these stack locations as separate storage cells of four types: parameter octets, parameter words, local-variable octets and local-variable words.

A word cell and its octet parts are connected by implied assignments. An assignment to an octet cell makes its containing word cell opaque. An assignment to a word cell also updates the octet cells within the word.

The program may access the same memory location both through an **SP**-relative access and in some other way, either using an absolute memory address or a register-indirect access with another base register. At present Bound-T assumes that such aliasing does not occur for loop-counting computations.

5.4 Condition codes and the CCR

The **CCR** condition flags that are modelled for analysis are **Z** (result zero) and **C** (carry or borrow). For arithmetic analysis Bound-T defines the flags only for an integer operation that can give a significant result for unsigned operands.

For example, the **ADD.B** instruction sets **Z** if the result is zero. If both operands are understood as unsigned values this can happen only if the addition overflows. Since Bound-T tracks only non-overflowing computation, its model of **ADD.B** generally sets **Z** to an opaque value. However, when the source operand of **ADD.B** is an immediate value in the range 128 .. 255 that is understood as a 2's complement negative number, Bound-T models the **Z** flag as set when the sum of the destination register and the negative immediate operand is zero.

Here are some examples:

- **ADD.B R3L, R4H** is modelled as **R4H** := **R4H** + **R3L**, **Z** := opaque, **C** := opaque.
- **ADD.B #255, R4H** is modelled as **R4H** := **R4H** – 1, **Z** := R4H = 1, C := **R4H** = 0.

This model is incomplete and rather ad-hoc. For example, in the real processor the instruction sequence

MOV.B #255, R3L

ADD.B R3L, R4H

has exactly the same effect on **R4**, **Z** and **C** as the single instruction **ADD.B #255, R4H** but in the Bound-T model the instruction sequence results in opaque values for **Z** and **C** while the single instruction sets **Z** and **C** to non-opaque values as shown above.

The following table shows how Bound-T models the flag setting by the H8/300 instructions (roughly alphabetically ordered). The table applies to instructions that stand alone, it does not apply when two 8-bit instructions are combined into a 16-bit operation as described in section 5.3. The symbol "–" means that the flag is not changed (neither in the model nor in the real processor). When a register (**Rd** or **Rs**) is used in the **Z** and **C** conditions the value of the register *before* the instruction is meant, and likewise for the **Z** and **C** flags themselves. The word "result" means the value computed by the instruction (eg. the value of Rd *after* the instruction).

Table 10: Condition flag definitions

Instruction	*Z condition*	*C condition*
ADD.B #<k = 0 .. 127>, Rd	opaque	opaque
ADD.B #<k = 128 .. 255>, Rd	**Rd** = 256 – **k**	**Rd** < 256 – **k**
ADD.B Rs, Rd **ADD.W Rs, Rd** **ADDX**	opaque	opaque
ADDS	–	–
AND	result = 0	–
ANDC #k, CCR	**Z** and (bit 2 of **k**)	**C** and (bit 0 of **k**)
BAND, **BIAND**, **BILD**, **BIOR**, **BIXOR**, **BLD**, **BOR**, **BXOR**	–	opaque
BCLR, **BNOT**, **BSET**, **BST**	–	–
BTST	opaque	–
Bcc	–	–
CMP.B Rd, Rs	**Rd** = **Rs**	**Rd** < **Rs**
DAA	opaque	opaque
DAS	opaque	–
DEC Rd	**Rd** = 1	–
DIVXU	opaque	opaque
EEPMOV	–	–
INC Rd	**Rd** = 255	–
JMP, **JSR**	–	–
LDC #k, CCR	bit 2 of **k**	bit 0 of **k**
LDC Rs, CCR	opaque	opaque
MOV, **MOVFPE**, **MOVTPE**	result = 0	–
MULXU	–	–

Instruction	*Z condition*	*C condition*
NEG Rd	**Rd** = 0	**Rd** > 0
NOT	opaque	–
OR	result = 0	–
ORC #k, CCR	**Z** or (bit 2 of **k**)	**C** or (bit 2 of **k**)
POP, **PUSH**	result = 0	–
ROTL, **ROTR**, **ROTXL**, **ROTXR**	opaque	opaque
RTE (see note below), **RTS**, **SLEEP**, **STC**	–	–
SUB.B Rd, Rs **SUB.W Rd, Rs**	**Rd** = **Rs**	**Rd** < **Rs**
SUBS	–	–
SUBX	opaque	opaque
XOR	**Rd** = **Rs**	–
XORC #k, CCR	**Z** xor (bit 2 of **k**)	**C** xor (bit 0 of **k**)

Note that **RTE** is modelled as having no effect on the **CCR**, although in fact **RTE** pops the **CCR** from the stack. The model corresponds to the fact that the entire execution of the exception handler that ends with **RTE** preserves the original value of the **CCR**.

5.5 Conditional branches and the "-bcc" Option

The conditional branch instruction **Bcc** supports 16 branch conditions which are logical combinations of the **CCR** condition flags **N** (negative), **Z** (zero) and **V** (overflow). These 16 codes fall into five groups:

- the two conditions for "always true" and "always false";
- the two conditions for equality or inequality;
- the four conditions that reflect comparison of unsigned integers;
- the four conditions that reflect the same comparisons of signed integers; and
- the four conditions for "plus" or "minus", and overflow or no overflow.

Since Bound-T only models unsigned computation on the H8/300, the condition codes for signed comparison are considered opaque by default. However, this will usually prevent Bound-T from finding bounds for loops that use signed counter types, such as the 'C' language "int" type, even if the counters only take on non-negative values and the loop could as well use the corresponding unsigned type such as the 'C' language "unsigned int" type. To work around this problem, the command-line option *-bcc=unsigned* makes Bound-T interpret the signed conditions as equivalent to the corresponding unsigned conditions. This option should be used only when the counter values stay in the non-negative signed range (0 .. 127 for 8-bit counters, 0 .. 32 767 for 16-bit counters).

The table below shows how Bound-T models the **Bcc** condition codes under the two possible values for the *-bcc* option. The option changes the interpretation only for the signed comparison conditions.

Table 11: Condition codes in Bcc

Mnemonic Bcc	*Meaning*	*Interpretation depending on the -bcc option*	
		-bcc=signed	*-bcc=unsigned*
BRA (BT)	Always (True)	True (branch)	Same
BRN (BF)	Never (False)	False (don't branch)	Same
The unsigned comparisons:			
BHI	High (>)	C = 0 and Z = 0	Same
BLS	Low or Same (≤)	C = 1 or Z = 1	Same
BCC (BHS)	High or Same (≥)	C = 0	Same
BCS (BLO)	Low (<)	C = 1	Same
The equality conditions:			
BNE	Not Equal (≠) or Not Zero	Z = 0	Same
BEQ	Equal (=) or Zero	Z = 1	Same
The overflow and sign conditions:			
BVC	No Overflow	Opaque	Opaque
BVS	Overflow	Opaque	Opaque
BPL	Plus (≥ 0)	Opaque	Opaque
BMI	Minus (< 0)	Opaque	Opaque
The signed comparisons:			
BGE	Greater or Equal (≥)	Opaque	As BHS: C = 0
BLT	Less Than (<)	Opaque	As BLO: C = 1
BGT	Greater Than (>)	Opaque	As BHI: C = 0 and Z = 0
BLE	Less or Equal (≤)	Opaque	As BLS: C = 1 or Z = 1

5.6 Dynamic jumps and calls

What they are

The H8/300 instructions **JMP** (jump) and **JSR** (jump to subroutine, call) can use a dynamically computed target address, in two forms: register indirect (**@Rn**), where the target address is in a 16-bit register, and memory indirect (**@@aa:8**) where the target address is in the memory word at a statically known address (**aa**) in the range 0 .. 255 (0 to FF hexadecimal). The analysis of such instructions is difficult because the possible target addresses must be deduced to build the control-flow graph (**JMP**) and call-graph (**JSR**).

Dynamic jumps are often used to implement "switch/case" code structures. Bound-T tries to support this form of dynamic jump as explained below. In general, Bound-T is not able to find the targets of dynamic calls by analysis; they must usually be listed in an assertion. However, for the H8/300 the form **JSR @@aa:8** is supported in a limited form as explained below.

Jump address tables and switch/case statements

When a switch/case statement has a dense (numerically consecutive) set of case labels the compiler often implements the statement with an indexed jump. The code contains a table with the addresses of the case branches. The table is indexed by the switch/case variable. In the H8/300 processor, the following instruction sequence seems to be used:

```
mov.w @(base, Ri), Rj
jmp @Rj
```

Here the constant displacement (base) is the address of the table, the register **Ri** contains the offset into the table as computed from the switch/case variable, and **Rj** (which may be the same register as **Ri**) contains the address of the chosen case branch.

Bound-T detects this code pattern and tries to find all the case branches by deriving an upper bound on the value of **Ri** at the first instruction (**mov.w**). A lower bound of zero is assumed. If a reasonable upper bound on **Ri** is found, it defines the full length of the address table (from base to base + upper bound - 1). Assuming that the table has a constant content, as loaded from the binary executable file, we can read out the addresses of all case branches, that is, all possible targets of the dynamic jump instruction (**jmp @Rj**). Bound-T uses this method to resolve the dynamic jump and complete the control-flow graph. It also emits a warning that it has assumed a constant address table.

Vectored (memory-indirect) jumps and calls

The memory-indirect dynamic call instruction **JSR @@aa:8** seems to occur frequently in H8/300 code. Bound-T supports it under the assumption that the memory word that holds the target address is constant. In other words we assume that the memory area 0 .. 255 contains constant "vectors" or pointers to frequently used subprograms and that the apparently "dynamic" call **JSR @@aa:8** is merely an abbreviation for the normal **JSR** which needs an additional instruction word for the 16-bit address. The similar assumption is taken for the memory-indirect jump instruction **JMP @@aa:8**.

If the **@@aa:8** operand refers to a vector address (**aa**) that is defined (loaded) by the memory image in the binary executable file, Bound-T reads the final target address from the memory image and models the **JSR** or **JMP** as a call or jump with this static target address.

Otherwise, that is if the vector address is not defined in the memory image, Bound-T concludes that the final target is dynamically computed. In this case, a **JSR** is modelled as a dynamic call that can be resolved by analysis or by an assertion. A **JMP** is modelled as a subprogram return point, with an error message to alert the user.

5.7 Memory configurations

Typical H8/300 chips have internal ROM (or PROM) and RAM but can also use external ROM and RAM.

External memory can be used *together* with internal memory when the chip is configured so that the internal and external memories use *different* regions of the address space.

External memory can be used *instead of* internal memory when the chip is configured so that the address regions that usually map to internal memory instead map to external memory.

Some addresses are mapped to the on-chip register field, and some address ranges may be unmappped (reserved) and should not be accessed.

The term *memory map* means the division of the 16-bit address space into internal address, external addresses, on-chip register addresses and unused address. The memory map depends on the type of the chip and the configuration of the chip.

For example, for the chips in the H8/3297 series, the chip type defines the amount of internal memory: from 16 to 60 kilo-octets or ROM and from 0.5 to 2 kilo-octets of RAM. The choice of internal or external memory depends on the two input pins that define the operation "mode" and on one bit in the System Configuration register (bit **RAME** in **SYSCR**).

The H8/3297 chips can operate in three modes:

- Mode 3 is called "single-chip mode". In mode 3, the chip uses only internal memories and large parts of the address space are reserved.
- Mode 2 is called "expanded mode with on-chip PROM". In mode 2 the chip can use external memory for those addresses that are reserved in mode 3, but still uses the internal ROM or PROM as in mode 3.
- Mode 1 is called "expanded mode without on-chip PROM". In mode 1 the address region for the internal ROM/PROM is mapped to external memory instead; the only accessible internal memory is the internal RAM (and the on-chip register field).

In modes 1 and 2, the target program can choose between internal or external RAM by controlling bit **RAME** in the **SYSCR** register. If **RAME** is off, the internal RAM addresses are mapped to external memory instead.

That was how the memory map is defined in the H8/3297 series of H8/300 chips. Other chip series may have other rules.

In general, the memory map determines two aspects of a memory access:

- how long it takes (longer for external memory), and
- which storage location is accessed (internal or external).

In modes 1 and 2 the target program can control the **RAME** bit to access either the internal RAM or an external memory. This means that the interpretation of an address in this region depends dynamically on the value of **RAME**, which could be hard to handle in a static analysis. In effect, there would be two memory "banks", an internal bank and an external bank, with **RAME** as the bank selector.

Bound-T does not support or detect dynamic changes to **RAME**. Instead Bound-T assumes that the *memory map is fixed* throughout the analysed part of the target program. For example, for chips in the H8/3297 series Bound-T accepts command-line options to specify the mode and the value of **RAME**.

If your target program does change the memory map dynamically, perhaps by changing **RAME**, you should split the analysis in a like way and analyse separately each part of the program with the corresponding memory-map options. If this is too difficult, you should use the worst-case memory-map options (all memory is external) but this may lead to a considerably over-estimated WCET bound.

Even if you can specify the correct memory map options, Bound-T may still over-estimate the time for those dynamic memory accesses that Bound-T cannot bound enough to decide if they map to internal or external memory. A common case is stack accesses; Bound-T usually does not know the absolute value of the stack pointer. Here you should use the command-line option *-stack* to tell Bound-T if the stack is in internal or external RAM.

5.8 Timing accuracy and approximations

Bound-T reports WCET values that take into account most of the timing features of the H8/300. This section explains these features, how Bound-T models them, and where Bound-T must make assumptions or approximations.

Memory wait states

Access to on-chip memory (ROM, RAM or register field) does not involve wait states, but off-chip memory access can force the processor to wait. The number of wait states must then be given as command-line options *-read_ws* and *-write_ws* for the reading and writing wait states respectively. By default Bound-T assumes zero wait states.

At present it is not possible to specify a different number of wait states for different parts (address ranges) of the off-chip memory.

Summary of approximations

The following table lists the cases where Bound-T uses an approximate model of the timing of H8/300 instructions.

Table 12: Approximations for instruction times

Case	*Description*	*Maximum Error*
Memory areas with different access time	For a memory access with a dynamically computed address (register indirect) Bound-T cannot always determine which kind of memory area is accessed (on-chip memory, on-chip register field, or external memory). Bound-T assumes the worst case (external memory).	Per octet access: 4 states plus the number of wait-states defined for external memory.
MOVFPE **MOVTPE**	The operand access time is variable between 9 and 16 states because these instructions synchronize with the peripheral E (enable) clock.. Bound-T assumes the worst case.	Per executed instruction: 7 states.

6 PROCEDURE CALLING PROTOCOLS

6.1 Calls and returns in the H8/300

In this chapter, we discuss how H8/300 programs use subprograms (procedures and functions) and explain how Bound-T identifies subprograms and analyses the control-flow and data-flow across subprogram calls and returns.

Subprograms, calls and returns are important here because Bound-T uses a modular analysis method in which each subprogram is first analysed separately and without assumptions on the actual parameter values. This intra-procedural analysis is fast and efficient when it succeeds. If it turns out that the subprogram cannot be fully analysed in this way – for example, if the value of a parameter defines the number of iterations of a loop – then Bound-T switches to an inter-procedural analysis in which it considers calling context: for each call to the subprogram, the subprogram is re-analysed in the context of the parameter values and global variable values that this call passes to the subprogram.

Hardware aspects of the calling protocol

The H8/300 instruction set contains two instructions specifically intended for subprogram calls: **BSR** (branch to subroutine) and **JSR** (jump to subroutine). **BSR** specifies the entry address of the called subprogram as a PC-relative static offset. **JSR** can use a 16-bit static entry address, a register-indirect address (**@Rn**) or a memory-indirect address (**@@aa:8**). The addressing modes are discussed in section 5.6. The present chapter discusses how parameters are passed from the caller to the callee and back, how the stack is used, and which registers can be changed or are preserved across the call. Rules for this are usually called a *procedure calling standard* or *calling protocol* or *calling convention*.

The H8/300 architecture defines only one aspect of the calling protocol: how the return address (and, for interrupts, the **CCR**) is managed on the stack. Namely:

- The call instruction (**BSR** or **JSR**) pushes the return address (**PC**) on the stack. The subprogram normally ends with the return instruction (**RTS**) which pops the return address from the stack and continues execution in the caller.
- When an interrupt occurs, the processor's interrupt mechanism pushes the condition code register (**CCR**) and the **PC** on the stack and then enters the interrupt handler subprogram. The handler subprogram normally ends with a return-from-exception instruction (**RTE**) which pops the **CCR** and the **PC** from the stack and lets execution resume at the interrupt point.

It is possible to implement mechanisms for subprogram calls and returns that use other H8/300 instructions, for example dynamic jumps (**JMP @Rn**). Bound-T only understands the native method that uses **BSR**, **JSR**, **RTS** and **RTE**.

Software aspects of the calling protocol

The H8/300 architecture does not define parameter passing nor saving and restoring of registers across calls. There does not seem to be a single standard for this aspect of H8/300 programming; different compilers use different rules, and assembly-language programmers are free to define their own rules.

Bound-T understands and supports the following calling protocols:

- The protocol used in the GNU compilers (gcc).
- The protocol used in the compilers from IAR Systems.

In the remaining sections of this chapter, we explain each supported calling protocol and how Bound-T interprets it. Note that a calling protocol usually contains some rules that Bound-T does not rely on for its analysis; thus we in fact support a superset of the calling protocol in which these irrelevant rules need not be followed. In particular, the rules that govern how a compiler chooses a parameter-passing mechanism for a given source-language parameter are usually not significant to Bound-T.

6.2 The GNU calling protocol

Introduction

This section explains how Bound-T supports the procedure calling protocol used by the GNU H8/300 C compiler, using the default compilation options. This section is based on the H8/300 GCC Application Binary Interface description [8] and on observation of compiled code.

The GCC calling protocol has the following features:

- The native instructions **BSR**/**JSR** and **RTS** are used.
- Parameters are passed in registers and (if many) on the stack.
- The callee can alter the values of registers **R0**, **R1**, **R2**, **R3**.
- The callee must not alter the values of registers **R4**, **R5**, **R6**, **R7** or must save the original value and restore it before returning to the caller.

Parameter passing

Up to three parameters are passed in **R0**, **R1** and **R2**. The rest of the parameters are passed on the stack, pushed there before the call. Parameter sizes on the stack are rounded to an even number of octets. For example, an octet parameter is held in one word of stack space.

Subprogram call

The subprogram call sequence consists of pushing the stack-based parameters on the stack, loading register-based parameters into registers, and executing a **BSR** or **JSR**.

Use of the stack in the callee

The callee subprogram usually allocates stack space for its local and temporary variables. It can do this by **PUSH** instructions or by decreasing the **SP** in some other way (eg. by subtraction). The callee uses the stack also to save the values of the callee-save registers **R4**-**R6** when necessary. The exact lay-out of the stack frame in the callee depends on the GCC version (see [8]) and is not relevant to Bound-T.

Of course, if the callee itself performs calls to other subprograms, it may use the stack for parameters to these other subprograms. This means that **SP** can vary quite dynamically during the execution of a subprogram.

Access to stacked parameters and locals

Under the GCC option *-fomit-frame-pointer* there is no frame pointer. This means that all stack-based data must be accessed using **SP**-relative addressing (register indirect based on **SP**). However, as the value of **SP** can vary during the execution of the subprogram, so must the offset (displacement) be varied. For example, if the address **SP** + 8 accesses a given local variable before a **PUSH** instruction, the same variable must be accessed with **SP** + 10 after the **PUSH** instruction decreases **SP** by two.

Bound-T tracks the changes in **SP** and translates **SP**-relative addresses with varying offsets to addresses relative to the **SP** on entry (after the **BSR** or **JSR**) with fixed offsets. The fixed offset for a parameter is 2 or more (offset 0 refers to the return address). The fixed offset for local variables is negative.

Bound-T does not yet support the use of frame pointers.

Return from subprogram

To return, the subprogram pops its local variables from the stack, restores the caller-save registers from the stack, and executes **RTS**.

There may be more than one such return point in a subprogram.

Library subprograms that violate the calling standard

Run-time libraries sometimes contain special subprograms that are meant to be called only by compiler-generated code and have non-standard calling sequences. No information is currently available on this question for the GCC libraries.

6.3 The IAR calling protocol

This section sketches how Bound-T supports the procedure calling protocol used by the IAR Systems H8/300 C compiler. This section is based on the compiler manual [7] and on observation of compiled code.

The IAR compiler supports the native 64-kilo-octet memory space but also supports *banked code* by which a certain address window can be mapped to one of several external memory banks containing code. The calling protocol becomes more complex for banked code. At present Bound-T does *not* support banked code and we discuss only the non-banked (native) IAR calling protocol here.

The non-banked IAR calling protocol has the following features:

- The native instructions **BSR**/**JSR** and **RTS** are used.
- The first parameter is normally passed in a register, using **R6L**, **R6** or **R5**:**R6** depending on the size of the parameter.
- Other parameters are normally passed on the stack, but the compiler option *-ur* can specify more registers for parameter passing.
- Normally a function must preserve all registers (except the registers used for passing parameters or the return value), but the compiler option *-uu* can specify registers that can be trashed.

The IAR compiler options *-ur* and *-uu* are supported. If they are set to non-default values, you must use the corresponding Bound-T command-line options to tell Bound-T which values were used in compilation.

7 WARNING AND ERROR MESSAGES

7.1 Warning messages

The following lists the Bound-T warning messages that are specific to the H8/300 or that have a specific interpretation for this processor. The messages are listed in alphabetical order. Any *italic* word or symbol in the message stands for a variable string.

The Bound-T Reference Manual [2] explains the generic warning messages, all of which may appear also when the H8/300 is the target. The Bound-T Assertion Language manual [3] explains the generic warning messages related to assertions. The specific warning messages listed below refer mainly to unsupported or approximated features of the H8/300 or to possible errors in the given executable program file.

As Bound-T evolves, the set and form of these messages may change, so this list may be out of date to some extent. However, we have tried to make the messages clear enough to be understood even without explanation. Feel free to ask Tidorum for an explanation of any Bound-T output that seems obscure.

Table 13: Warning messages

Warning Message		*Meaning and Remedy*
Assignment to *R* makes stack height unknown	*Reasons*	The instruction assigns a new value to the 8-bit register *R* which is either **R7L** or **R7H** which means that Bound-T's model loses track of how much has been pushed on the stack, that is, the local stack height becomes unknown. This means that Bound-T may be unable to track how data flows in parameters and local variables. It also means that Bound-T will not be able to bound the stack usage of the subprogram.
	Action	Study why this instruction exists and try to remove it or replace it with an instruction that assigns to the word register **R7**.
Assuming constant jump-address table at *A*	*Reasons*	Bound-T has detected an instruction pair of the form **MOV.W @(A, Ri), Rj** **JMP @Rj** and assumes that this pair uses a constant table of code addresses starting at the address *A*. See section 5.6.
	Action	Check that the assumption holds. If the table in fact is not constant, either make it constant (justifying the assumption) or change the instruction pair somehow to prevent Bound-T from detecting it (for example, add a **NOP** instruction between the **MOV.W** and the **JMP**).
Assuming constant vector to *A* at *V*	*Reasons*	Bound-T has found an instruction of the form **JMP @@V** or **JSR @@V** and has also found that the memory image in the binary executable file loads the address *A* into the vector slot *V*. Bound-T assumes that the vector slot will be held constant (always contain *A*) and therefore models the instruction as a jump to *A*, or a call of the subprogram with address *A*, respectively. See section 5.6.
	Action	Check the target program and verify that the assumption holds. If the assumption is wrong, change the target program.

Warning Message	*Meaning and Remedy*	
COFF ".file" entry with no file-name.	*Reasons*	The COFF symbol table contains a primary entry of ".file" type but it is not followed by an auxiliary file-name entry.
	Action	Supply a correct COFF file for the target program.
COFF line-number table refers to symbol index *N* which has no subprogram scope	*Reasons*	The COFF table that connects source-code line numbers with code addresses identifies a source line that does not seem to be located in a subprogram.
	Action	This is probably harmless and no action is needed.
Disabled internal RAM implies -stack=external	*Reasons*	The command-line option *-internal_ram=disabled* overrides the option (or default setting) *-stack=internal*. If the internal RAM is not used, the stack must be in external RAM.
	Action	Check the command-line options and correct if necessary.
Displacement *D* taken as signed = -*S*	*Reasons*	The 16-bit displacement *D* in a register-indirect operand is large enough (at least 32 768 or 8000 hex) to be interpreted as a negative (2's complement) offset *S*.
	Action	None. Even if the negative interpretation is wrong, Bound-T will apply 16-bit wrap-around to the final address which will compensate.
Fetching code from on-chip register field at *A*	*Reasons*	The immediate reason for this warning is that the control-flow analysis has given the Program Counter (**PC**) an address *A* that points into the on-chip register field. As the on-chip registers are usually peripheral control registers, this is an unusual situation (but perhaps not illegal). It may mean that a dynamic jump or call has been misanalysed; the maximum value of a switch/case index may be over-estimated or the assumption of a constant address table is wrong. An alternative error is that the wrong device was chosen, one where the on-chip register field covers some addresses that contain code in the real target device.
	Action	See section 5.6 and check the program. Check that the *-device* option is correct for this program.
Fetching from odd address *A*	*Reasons*	The immediate reason for this warning is that the control-flow analysis has given the odd value *A* to the Program Counter (**PC**). As instruction words always have an even octet address this is an unusual situation (but not illegal, the processor ignores the least significant **PC** bit). It may mean that a dynamic jump or call has been misanalysed; the maximum value of a switch-case index may be overesti-mated or the assumption of a constant address table is wrong.
	Action	See section 5.6 and check the program
Immediate operand *N* taken as signed = -*S*	*Reasons*	The immediate 8- or 16-bit operand *N* is large enough (at least 128 or 32 768) to be interpreted as a negative (2's complement) number *S*.
	Action	None. However, the interpretation may be wrong and may prevent the analysis of loop bounds.
Invalid COFF symbol section number: *symbol*: *section*	*Reasons*	The COFF symbol table gives a *section* number for this *symbol* that is not a valid section number in this COFF file.
	Action	Supply a correct COFF file for the target program.

Warning Message		*Meaning and Remedy*
Meaning of *P* not understood and thus ignored	*Reasons*	The assertion *P* on the value of the property *bcc_signed* allows more than one value, or a value that is not 0 or 1. Only the values 0 and 1 have meaning for this property.
	Action	Correct the assertion. See section 3.6.
Mode 3 implies -internal_ram=enabled	*Reasons*	The command-line option *-mode=3* overrides the option *-internal_ram=disabled*. If there is no external memory, the internal memory must be used. See section 2.2.
	Action	Check the command-line options and correct if necessary.
Mode 3 implies -stack=internal	*Reasons*	The command-line option *-mode=3* overrides the option *-stack=extermal*. If there is no external memory, the stack must be in internal memory. See section 2.2.
	Action	Check the command-line options and correct if necessary.
Reading word from odd address *A*	*Reasons*	The current instruction reads a (constant) word from the odd address *A*. This is unusal because word addresses are usually even. Perhaps the control-flow analysis is exploring an invalid path and is interpreting constant data as instructions.
	Action	Check that the instruction is correct. Perhaps alter the program.
Resolved callee address *A* is not a valid code address. This dynamic call is not resolved further.	*Reasons*	Bound-T has analysed the possible values of the target address in a dynamic call instruction (a **JSR** with a register or indirect operand) but the result *A* is not a valid code (instruction) address because it is too large or is an odd number. Either Bound-T is analysing memory content that is not meant to be executed, or the analysis is wrong, for example due to aliasing, or the target program is in error.
	Action	Check that this instruction (**JSR**) and the computation of the target address are correct. Find out which subprogram or subprograms are meant to be called and use an assertion to specify these subprograms as the possible callees for this dynamic call.
Skipped misplaced COFF auxiliary symbol *kind*	*Reasons*	The COFF symbol table contains an "auxiliary" symbol entry of this *kind* in an unexpected place. The entry is skipped.
	Action	Supply a correct COFF file for the target program.
Skipping COFF symbol, Kind = *kind*, Nature = *nature*	*Reasons*	A COFF symbol entry of the named *kind* and *nature* was found in an unexpected place.
	Action	Supply a correct COFF file for the target program.
Target of vector at *V* is unknown	*Reasons*	Bound-T has found an instruction of the form **JMP @@V** or **JSR @@V** but the memory image in the binary executable file does not load anything into the vector slot *V*. Bound-T assumes that the vector slot will be defined dynamically at execution time.
	Action	See section 5.6.
The COFF symbol-table is empty	*Reasons*	No symbol table, or an empty symbol table was found in the COFF file.

Warning Message		*Meaning and Remedy*
	Action	If you want to use symbolic names and/or source-code references, supply a COFF file with a symbol table. The usual solution is to recompile and relink the target program with "debug" options, for example *-g* for the GNU compiler.
Unexpected length *N* of optional file header (skipped).	*Reasons*	The COFF optional file header structure has a different length (*N*) than Bound-T expects. This probably means that the header structure is different from Bound-T's expectation, so the whole optional header is skipped.
	Action	Usually none. If the information in the optional header is essential for your analysis, please ask Tidorum for help.
Unresolved JSR @Rn/@@aa taken as no-operation.	*Reasons*	Bound-T was not able to find a single possible value for the register operand (**Rn**), or the indirect memory vector at address **aa**, in a dynamic call instruction, and therefore cannot analyse the callees.
	Action	Find out which subprogram or subprograms are meant to be called and use an assertion to specify these subprograms as the possible callees for this dynamic call.
Unsure about COFF symbol with Storage Class = *class*: *symbol*	*Reasons*	The COFF symbol-table gives the *symbol* an unexpected or unknown storage *class* so that Bound-T cannot deduce the nature of the object that the *symbol* denotes. The COFF file may be incorrect or may use a different variant of COFF than Bound-T expects.
	Action	Usually none. If the nature of the *symbol* is essential for your analysis, please ask Tidorum for help.
WCET omits sleeping time for SLEEP instruction	*Reasons*	The program contains a **SLEEP** instruction at this point.
	Action	Understand that the WCET bound given for this subprogram does not include the time spent sleeping.

7.2 Error messages

The following table lists the Bound-T error messages that are specific to the H8/300 or that have a specific interpretation for this processor. The messages are listed in alphabetical order. Any *italic* word or symbol in the message stands for a variable string.

The Bound-T Reference Manual [2] explains the generic error messages, all of which may appear also when the H8/300 is the target. The Bound-T Assertion Language manual [3] explains the generic error messages related to assertions.

As Bound-T evolves, the set and form of these messages may change, so this list may be out of date to some extent. However, we have tried to make the messages clear enough to be understood even without explanation. Feel free to ask Tidorum for an explanation of any Bound-T output that seems obscure.

Table 14: Error messages

Error Message		*Meaning and Remedy*
Access to reserved memory address *A* is assumed to reach external memory	*Problem*	The target program accesses a memory address *A* that is defined as "reserved" in the memory map of the chosen H8/300 device. The real behaviour is undefined; Bound-T assumes that external memory is accessed (worst case for execution time).
	Reasons	Programming error, or wrong *-device* option chosen, or misanalysed dynamic memory address.
	Solution	Check that the *-device* option names the correct device.
A line of length *N* characters cannot be an S-record.	*Problem*	The S-record file (the target program file, or a file named in a command-line option *-srec=file*) has a line with a length (*N* characters) that is impossible for a well-formed S-record. This error message is followed by another that gives the number of the line that contains the error.
	Reasons	Perhaps the file is not an S-record file, or is damaged, or uses an S-record format variant that Bound-T does not support.
	Solution	Obtain an S-record file in the form that Bound-T supports.
A line of length *N* characters cannot be an S-record with *K* octets.	*Problem*	The S-record file (the target program file, or a file named in a command-line option *-srec=file*) has a line with a length (*N* characters) and a *Record length* field (*K* octets) that are incompatible. This error message is followed by another that gives the number of the line that contains the error.
	Reasons	Perhaps the file is not an S-record file, or is damaged, or uses an S-record format variant that Bound-T does not support.
	Solution	Obtain an S-record file in the form that Bound-T supports. See section 2.4.
A line that starts with "*C*" cannot be an S-record.	*Problem*	The S-record file (the target program file, or a file named in a command-line option *-srec=file*) has a line that starts with the two-character string *C* which is impossible for an S-record (as here defined). This error message is followed by another that gives the number of the line that contains the error.
	Reasons	Perhaps the file is not an S-record file, or is damaged, or uses an S-record format variant that Bound-T does not support.
	Solution	Obtain an S-record file in the form that Bound-T supports. See section 2.4.
At most one extra-symbols file allowed; it was "*file1* ". The file "*file2* " is rejected.	*Problem*	The command-line contains more than one *-sym=file* option. The first such option named *file1*; the current option names *file2*.
	Reasons	The command line is in error.
	Solution	Use this option at most once per command.

Error Message		***Meaning and Remedy***
At most one S-record file allowed; it was "*file1* ". The file "*file2* " is rejected.	*Problem*	The command-line contains more than one *-srec=file* option. The first such option named *file1*; the current option names *file2*.
	Reasons	The command line is in error.
	Solution	Use this option at most once per command.
Cannot determine executable file type	*Problem*	Bound-T tried to determine the type (COFF, S-record, or UBROF) of the given executable file, but failed.
	Reasons	The file may be of a type unknown to Bound-T, or it may be a unknown variant of a known type.
	Solution	Check that the correct file was named on the command line. If necessary, use the option *-coff* , *-srec*, or *-ubrof* to force Bound-T to use the right file type.
Cannot open the S-record file "*filename*".	*Problem*	Bound-T could not open and read the S-record file given as the target program executable file, or named in an *-srec=filename* command-line option.
	Reasons	There is no file with this name (*filename*) or the file is protected against reading.
	Solution	Correct the command-line (*filename*) or change the protection of the file to allow reading.
Cannot read file	*Problem*	The target program executable file, named on the command line, exists but cannot be read (error while reading).
	Reasons	You may not have the right to read this file.
	Solution	Correct the access rights of the file.
COFF block has no ".eb" symbol	*Problem*	The current block in this COFF file is not ended by a C_Block bracket of type ".eb" (end block).
	Reasons	The COFF file is damaged or uses a variant of COFF that Bound-T does not support.
	Solution	If this problem has some serious effects, obtain a correct COFF file in a form that Bound-T supports.
COFF function has no ".ef" symbol	*Problem*	The current function (subprogram) in this COFF file is not ended by a C_Fcn bracket of type ".ef" (end function).
	Reasons	The COFF file is damaged or uses a variant of COFF that Bound-T does not support.
	Solution	If this problem has some serious effects, obtain a correct COFF file in a form that Bound-T supports.
COFF symbol *S* has *N* unexpected auxiliary symbols; skipped	*Problem*	The COFF symbol at index *S* has *N* (one or more) attached or "auxiliary" symbols that modify the meaning or properties of the primary symbol *S*. However, the primary symbol *S* is not of a kind that can have auxiliary symbols.
	Reasons	The COFF file is damaged or uses a variant of COFF that Bound-T does not support.

Error Message		*Meaning and Remedy*
	Solution	If this problem has some serious effects, obtain a correct COFF file in a form that Bound-T supports, or translate the target program to some other format (S-record or UBROF).
Computed S-record checksum *S* differs from record value *C*	*Problem*	The S-record file (the target program file, or a file named in a command-line option *-srec=file*) has a line where the Checksum field value *C* is different from the checksum *S* that Bound-T computes from the rest of the data on the line. This error message is followed by another that gives the number of the line that contains the error.
	Reasons	Perhaps the file is not an S-record file, or is damaged, or uses an S-record format variant that Bound-T does not support.
	Solution	Obtain an S-record file in the form that Bound-T supports. See section 2.4.
Could not define the symbol "*symbol*" at "*address*".	*Problem*	The additional symbol-table file named in a *-sym* command-line option contains a syntax error at this point. The error message shows the file name and the line number in the file.
	Reasons	The address field is probably in error; perhaps hexa-decimal digits are used without 16#...# around it.
	Solution	Correct the file.
Could not open the extra-symbols file "*filename*".	*Problem*	Bound-T could not open and read the additional symbol-table file named in a *-sym* command-line option.
	Reasons	There is no file with this name (*filename*) or the file is protected against reading.
	Solution	Correct the command-line (*filename*) or change the protection of the file to allow reading.
File not found	*Problem*	The target program executable file, named on the command line, does not exist.
	Reasons	There is a mistake on the command line.
	Solution	Correct the command line.
Function symbol at index *F* is not followed by a .bf entry *or* Function symbol at index *F* has no auxiliary "Begin Function" entry	*Problem*	In this COFF file, the symbol at index *F* identifies a function (subprogram), which means that the next symbol should be a C_Fcn "begin function" bracket, but it is not; or that there should be a "Begin Function" auxiliary symbol, but there is none.
	Reasons	The COFF file is damaged or uses a variant of COFF that Bound-T does not support.
	Solution	If this problem has some serious effects, obtain a correct COFF file in a form that Bound-T supports.
Invalid instruction	*Problem*	The program contains an instruction word or word pair that does not encode a valid H8/300 instruction.

Error Message		*Meaning and Remedy*
	Reasons	1. The executable file may be invalid. It may be damaged or contain a program for another member of the Renesas H8 processor family with an extended instruction set. 2. The entry address specified for the current subprogram may be invalid; it may point to data rather than code, or to code that is not loaded directly from the executable file but is created or copied here at run time. 3. The control-flow analysis may be exploring an impossible path. Perhaps a dynamic jump was misanalysed; perhaps the maximum value of a switch/case index was overestimated.
	Solution	Ensure that the executable file is correct and valid for the H8/300. Analyse only code that exists in the executable memory image at load time. Check the analysis of dynamic jumps.
Jump via dynamic vector taken as return	*Problem*	The program contains a memory-indirect dynamic jump instruction of the type **JMP @@aa:8** that Bound-T cannot resolve. See section 5.6. Bound-T treats this instruction as if it were a return instruction.
	Reasons	The programmer or compiler generated this instruction for some reason.
	Solution	Try to modify the program to avoid this instruction. If a dynamic jump of this kind is necessary, determine the possible jump destinations, analyse them with Bound-T as separate subprograms, and add their WCET bound to the WCET bound for the (truncated) subprogram that contains the dynamic jump.
Jump via register taken as return	*Problem*	The program contains a register-indirect dynamic jump instruction of the type **JMP @Rn** that does not seem to use an address table. See section 5.6. Bound-T treats this instruction as if it were a return instruction.
	Reasons	The programmer or compiler generated this instruction for some reason. Perhaps it is a C program that uses function pointers.
	Solution	Try to modify the program to avoid this instruction. If a dynamic jump of this kind is necessary, determine the possible jump destinations, analyse them with Bound-T as separate subprograms, and add their WCET bound to the WCET bound for the (truncated) subprogram that contains the dynamic jump.
Line numbers unknown for this subprogram	*Problem*	The COFF file defines no "first source line number" for the present subprogram.
	Reasons	Perhaps the subprogram was compiled without debugging information.
	Solution	Recompile with debugging information.
Line too long (over *maximum* characters) in the extra-symbols file	*Problem*	The additional symbol-table file named in a *-sym* command-line option contains a line that is longer than the maximum length that Bound-T supports.

Error Message		*Meaning and Remedy*
	Reasons	The line is too long. Note that blanks and other white-space characters are included in the length.
	Solution	Shorten the line.
No -device was specified	*Problem*	The command line lacks the option *-device=X* to choose the H8/300 device (chip).
	Reasons	Missing obligatory option.
	Solution	Add the option *-device=X* to the command line, with some valid device name *X*.
No instruction loaded at this address	*Problem*	The control-flow analysis has given the Program Counter (**PC**) a value that points to a memory location where nothing is loaded; the executable file does not contain data or code for this address.
	Reasons	See the error message "Invalid instruction".
	Solution	See the error message "Invalid instruction".
Option argument is not a number: "*value*"	*Problem*	In a command-line option of the form *-keyword=value*, the *value* is expected to be a number but the actual *value* is not a number.
	Reasons	Error in the command line.
	Solution	Correct the command line.
Parameter reference exceeds caller's frame	*Problem*	Bound-T has identified an **SP**-relative, register-indirect operand as a reference to a parameter (not a local). However, the parameter's offset relative to the **SP** on entry to this subprogram is so large that the parameter cannot reside in the caller's stack frame. This violates the calling protocol that Bound-T understands.
	Reasons	Unknown. Perhaps the program uses a novel calling protocol, or perhaps Bound-T has computed the local stack height incorrectly in the caller or the callee.
	Solution	Please report this problem to Tidorum for analysis.
Patching is not implemented for this target	*Problem*	The command line uses the generic option *-patch* to specify a patch file. However, patching is not implemented for the H8/300.
	Reasons	Generic option not implemented for this target.
	Solution	Remove the -patch option. If you need patching for the H8/300, ask Tidorum to implement it.
Return by offset *X* from stange state *S*	*Problem*	In this call, the callee is asserted to return to a point that is offset by *X* octets from the normal return point as offered by the caller, but the control-flow state at the normal return point is the unexpected state *S*.
	Reasons	Probably an error in Bound-T-
	Solution	Please report the problem to Tidorum.

Error Message		Meaning and Remedy
S-record too long; at most *N* characters allowed.	*Problem*	The S-record file (the target program file, or a file named in a command-line option *-srec=file*) has a line with a length that is longer than the maximum (*N* characters) that Bound-T supports. This error message is followed by another that gives the number of the line that contains the error.
	Reasons	Perhaps the file is not an S-record file, or is damaged, or uses an S-record format variant that Bound-T does not support.
	Solution	Obtain an S-record file in the form that Bound-T supports. See section 2.4.
Syntax error in extra-symbol line:*text*	*Problem*	The additional symbol-table file named in a *-sym* command-line option contains a syntax error at this point. The error message shows the file name, the line number in the file and the *text* of the line.
	Reasons	There are not exactly two blank-separated strings on the line.
	Solution	Correct the file.
The callee cell *C* points beyond the caller's frame which has *N* octets.	*Problem*	The subprogram under analysis seems to access a stacked parameter *C* that is not in the stack frame of the calling subprogram (the stack offset is larger than the frame size, *N*).
	Reasons	Unknown. Perhaps the program uses a novel calling protocol, or perhaps Bound-T has computed the local stack height incorrectly in the caller or the callee.
	Solution	Please report this problem to Tidorum for analysis.
The characters "*cccc*" are not a *B*-bit hexadecimal number.	*Problem*	The S-record file (the target program file, or a file named in a command-line option *-srec=file*) has the characters *cccc* in a field where only hexadecimal digits are expected, to form a *B*-bit number. This error message is followed by another that gives the number of the S-record that contains the error.
	Reasons	Perhaps the file is not an S-record file, or is damaged, or uses an S-record format variant that Bound-T does not support.
	Solution	Obtain an S-record file in the form that Bound-T supports. See section 2.4.
The error is in S-record number *N*.	*Problem*	This is an addition to some immediately preceding error message that reports an error in an S-record file. This line reports the number of the S-record (line number) that contains the error.
	Reasons	See the immediately preceding error message.
	Solution	See the immediately preceding error message.
Undefined instruction code	*Problem*	The current binary instruction, as fetched from the memory image of the target program, is not the encoding of a valid H8/300 instruction.
	Reasons	See the error message "Invalid instruction".

Error Message		*Meaning and Remedy*
	Solution	See the error message "Invalid instruction".
Unexpected end of COFF file	*Problem*	The COFF executable file is not complete.
	Reasons	The COFF format is inconsistent.
	Solution	Obtain a correct COFF file.
Unexpected end of file	*Problem*	The target program executable file is not complete.
	Reasons	The executable file format is inconsistent.
	Solution	Obtain a correct executable file.
Unexpected end of UBROF file	*Problem*	The UBROF executable file is not complete.
	Reasons	The UBROF format is inconsistent.
	Solution	Obtain a correct UBROF file.
Unknown COFF C_Block symbol : *S*	*Problem*	The COFF records of type C_Block are normally used to bracket the records for one scope block between C_Block records called ".bb" (begin block) and ".eb" (end block). The present C_Block record has a symbol *S* that is neither ".bb" or ".eb". Bound-T treats it as an ".eb" record.
	Reasons	The COFF file is damaged or uses a variant of COFF that Bound-T does not support.
	Solution	If this problem has some serious effects, obtain a correct COFF file in a form that Bound-T supports, or translate the target program to some other format (S-record or UBROF).
Unknown COFF C_Fcn symbol : *S*	*Problem*	The COFF records of type C_Fcn are normally used to bracket the records for one subprogram between C_Fcn records called ".bf" (begin function) and ".ef" (end function). The present C_Fcn record has a symbol *S* that is neither ".bf" or ".ef". Bound-T treats it as an ".ef" record.
	Reasons	The COFF file is damaged or uses a variant of COFF that Bound-T does not support.
	Solution	If this problem has some serious effects, obtain a correct COFF file in a form that Bound-T supports, or translate the target program to some other format (S-record or UBROF).
Unknown value for -bcc: *value*	*Problem*	The *value* given for the command-line option *-bcc=value* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line. Change *value* to *signed* or *unsigned.*
Unknown value for -internal_ram: *value*	*Problem*	The *value* given for the command-line option *-internal_ram=value* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line. Change *value* to *enabled* or *disabled.*

Error Message		*Meaning and Remedy*
Unknown value for -mode: *value*	*Problem*	The *value* given for the command-line option *-mode=value* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line. Change *value* to 1, 2 or 3.
Unknown value for -stack: *value*	*Problem*	The *value* given for the command-line option *-stack=value* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line. Change *value* to *internal* or *external*.

Tidorum Ltd

Tiirasaarentie 32
FI-00200 Helsinki
Finland

www.tidorum.fi
info@tidorum.fi
Tel. +358 (0) 40 563 9186
Fax +358 (0) 42 563 9186
VAT FI 18688130