Bound-T time and stack analyzer

Application Note

Atmel AVR

TR-AN-AVR-001
Version 1
2010-02-14
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.

Copyright © 2009, 2010 Tidorum Ltd.
This document can be copied and distributed freely, in any format, provided that it is kept entire, with no deletions, insertions or changes, and that this copyright notice is included, prominently displayed, and made applicable to all copies.

Document reference:	TR-AN-AVR-001
Document issue:	Version 1
Document issue date:	2010-02-14
Bound-T/AVR version:	4a2
Last change included:	BT-CH-0219
Web location:	*http://www.bound-t.com/app-notes/an-avr.pdf*

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

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 address *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, telefax or ordinary mail to the address given below.

Please note that our office is located in the time-zone GMT + 2 hours, and office hours are 9:00 -16:00 local time.

Cordially,

Tidorum Ltd.

Telephone: +358 (0) 40 563 9186
Fax: +358 (0) 42 563 9186
Web: *http://www.tidorum.fi/*
http://www.bound-t.com/
Mail: *info@tidorum.fi*

Post: 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. Some versions of Bound-T emit XML data with the *XML_EZ_Out* package written by Marc Criley at McKae Technologies.

Contents

Index of 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 by means of a static analysis of the machine code of the program. There are different versions of Bound-T for different target processors. This Application Note supplements the general Bound-T manuals (references [1] and [2]) by giving additional information and advice on using Bound-T for one particular target processor, the processor architecture known as the Atmel AVR [4]. This information includes:

- the kinds of input files (executable programs) that Bound-T for AVR can read,
- the AVR devices (chips, models) that Bound-T for AVR supports,
- the cross-compilers that Bound-T for AVR supports,
- the AVR-specific command-line options for Bound-T,
- the AVR-specific details of the Bound-T assertion language, and
- the AVR-specific warning and error messages that Bound-T can emit.

Furthermore, the Application Note details how the analysis in Bound-T handles the features of the AVR architecture, with emphasis on features for which the analysis is approximate or even absent.

There may be other Bound-T Application Notes on issues that are not limited to the AVR, but nevertheless can be relevant when using Bound-T on AVR programs. For example, there may be Application Notes dealing with the target-independent properties of certain cross-compilers, or the target-independent aspects of how Bound-T reads and interprets certain executable-program formats. Check the Bound-T web-site *http://www.bound-t.com/* for such information.

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 Reference Manual [1] and the Bound-T User Guide [2]. The User Guide contains a glossary of terms, many of which will be used in this Application Note.

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 the program under analysis contains complex loops that Bound-T cannot analyse automatically the user must set the repetition bounds for these loops. This is done by writing *assertions* in the Bound-T assertion language. Assertions can also guide and help the analysis in other ways.

This Application Note explains how Bound-T has been adapted to the architecture of the AVR processor and how to use Bound-T to analyse programs for this processor. To make full use of this information, the reader should be familiar with the register set 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 explains those Bound-T command arguments and options that are wholly specific to the AVR or that have a specific interpretation for this processor.
- Chapter 3 explains how to write assertions to guide the analysis of AVR program. This extends the Bound-T Assertion Language manual [3] with AVR-specific details.

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

- Chapter 4 describes the main features of the AVR architecture and how they relate to the functions of Bound-T.
- Chapter 5 defines in detail the set of AVR instructions and registers that is supported by Bound-T.
- Chapter 6 presents the supported cross-compilers and explains the procedure calling standards (conventions, protocols) that Bound-T supports.
- Chapter 7 lists all the AVR-specific warning and error messages that Bound-T may emit, explains what the messages mean and what the underluying problem may be, and suggests some ways to correct these problems.

1.3 References

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

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

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

[4] 8-bit AVR Instruction Set.
Atmel Corporation 2002, ref. 0856D AVR 08/02.

[5] AVR-Libc.
http://www.nongnu.org/avr-libc/user-manual/.

[6] ICC V7 for AVR – C Cross Compiler for the Atmel AVR.
ImageCraft Creations Inc., December 3, 2006.

[7] AVR COFF (Common Object File Format) Specification. Preliminary release.
Jo Inge Lamo, version 1.0, January 5, 1998.

[8] AVR IAR C/EC++ Compiler Reference Guide for Atmel Corporation's AVR Microcontroller.
IAR Systems, February 2005 (fourth edition), Part number CAVR-4.

[9] Analysing Switch-Case Tables by Partial Evaluation.
N. Holsti, Tidorum Ltd. Presented at the 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007), Pisa, Italy, July 3, 2007.
http://www.tidorum.fi/bound-t/reports/wcet2007/abstract.html.

1.4 Abbreviations and acronyms

See also reference [2] for terms specific to Bound-T and reference [4] for the mnemonic operation codes and register names of the AVR.

COFF	Common Object File Format [7]
ELF	Executable and Linking Format
IAR	The C/EC++ compiler from IAR Systems [8]
ICCV7	The C cross-compiler from ImageCraft [6]
RISC	Reduced Instruction Set Computer
SREG	Status Register.
TBA	To Be Added
TBC	To Be Confirmed
TBD	To Be Determined
UBROF	Universal Binary Relocatable Object Format
V	Overflow flag (in the SREG)
WCET	Worst-Case Execution Time
X	The X register, formed by the register pair r27:r26
Y	The Y register, formed by the register pair r29:r28
Z	1. The Z register, formed by the register pair r31:r30 2. The zero flag (in the SREG)

1.5 Typographic conventions

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

register	The name of an AVR register embedded in prose.
instruction	An AVR instruction.
-option	A command-line option for Bound-T or other tools.
symbol	A mathematical symbol or variable.
`text`	Text quoted from a text / source file or command.
identifier	An identifier from a program.

2 USING BOUND-T FOR AVR

2.1 Overview

This chapter begins the “user guide” part of this Application Note. It starts by giving an overview of the AVR features and tools that Bound-T currently supports and continues by describing the input formats and listing and explaining all AVR-specific command-line options.

2.2 Support overview

Table 1 below shows a summary of the AVR features and tools that Bound-T supports at present. Note that support for a particular cross-compilers, such as the IAR compiler, means only that Bound-T/AVR has some knowledge of this compiler; it does not mean that Bound-T/AVR can analyse all programs compiled by this compiler, and likewise for procedure calling standards.

Table 1: Supported AVR features, tools, formats

Features	Supported	Notes
Architecture and instruction set	Full AVR [4].	Some limitations for code memories over 64 KiB.
Devices	Most AVR devices. If in doubt, ask Tidorum to verify for your device.	Some limitations for code memories over 64 KiB.
Cross-compilers	IAR	Well supported, including some analysis of C++ virtual calls.
	ImageCraft ICCV7	
	GNU *gcc*	Poorly supported at present, because of *gcc's* complex use of the hardware stack (**SP** stack) when 16 bits wide. An 8-bit **SP** is better supported.
Procedure calling standards	ImageCraft ICCV7	With or without *-r20_23*.
	IAR	Twovariants supported.
	GNU *gcc*	Uses no "software" stack, only **SP**. Problematic for 16-bit **SP**.
Stacks	The normal **SP** stack and optionally a software-defined stack using the **X**, **Y**, or **Z** register as stack pointer.	The **SP** stack is poorly supported for *gcc* with a 16-bit **SP**.
Executable file formats	ELF with DWARF2 or DWARF3	
	UBROF 10 from IAR	
	COFF	
Execution-time unit	Processor clock cycle [4]	
Stack-space unit	Octet (8-bit byte)	

2.3 Input formats

Executable target-program files

The target program executable file can be supplied in three formats: standard ELF, standard COFF, or the proprietary UBROF format defined by IAR Systems. Bound-T can usually detect the actual file format automatically, but it can also be chosen with command-line options as explained later in this chapter.

The quantity and detail of the symbolic (debugging) information differs between the three formats. In particular, only the UBROF format includes information about virtual function calls, thus Bound-T can analyse such calls (find the set of possible callees) only for UBROF programs. If an ELF or COFF program contains such calls you must use assertions to tell Bound-T about the possible callees for each such call.

Patch files not supported

Bound-T provides the general 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 AVR version of Bound-T does not currently support patching and so no patch-file format is defined.

2.4 Command arguments and options

Generic options and arguments

The generic Bound-T command format, options and arguments are explained in the Bound-T Reference Manual [1] and apply without modification to the AVR version of Bound-T. The command line usually has the form

```
boundt_avr  options  target-program-file  root-subprogram-names
```

For example, to analyse the execution time on the AVR device ATmega64 of the main sub-program in the target program stored as the ELF file prog.elf under the option *-trace calls*, the command line is

```
boundt_avr  -device atmega64  -trace calls  prog.elf  main
```

Root subprograms can be named by the link identifier, if present in the program symbol-table, or by the entry address in hexadecimal form. Thus, if the entry address of the main subprogram is 12A0 (hex), the above command can also be given as

```
boundt_avr  -device atmega64  -trace calls  prog.elf  12A0
```

All the generic Bound-T options apply. There are additional AVR-specific options as explained below. The generic option *-help* makes Bound-T list all its options, including the target-specific options.

AVR-specific options

The additional AVR-specific options are explained in Table 2 below. Note that a target-specific option must be written as one string with no embedded blanks, so the option-name and its numeric or symbolic parameter, if any, are contiguous and separated only by the equal sign (=) but not by white space. For example, the form "*-format=elf*" is correct, "*-format = elf*" is not. In addition to the options in Table 2 there are some options specific to certain executable file formats; these are listed separately in Table 4.

The *-device* option is a general Bound-T option, but its values – the device names – are target-specific. Section 2.5 lists the presently supported AVR devices and discusses any restrictions or device-specific options.

Table 2: Command Options for AVR

Option		*Meaning and default value*
-device=<name> or just *-<name>*	*Function*	Choose the AVR target device (chip, model) by giving the *name* of the device. The presently supported AVR devices are listed in Table 7 below. The device *name* is case-insensitive. The equal sign is optional and the option can be written *-device <name>*.
	Default	There is no default; a device must be selected.
-endian=big	*Function*	For modelling 16-bit computations, assume that 16-bit numbers are stored in memory in big-endian order: more significant octet first (at address *A*, say), less significant octet second (at address *A*+1).
	Default	The default is *-endian=little*.
-endian=little	*Function*	For modelling 16-bit computations, assume that 16-bit numbers are stored in memory in little-endian order: less significant octet first (at address *A*, say), more significant octet second (at address *A*+1).
	Default	This is the default.
-format=<form> or just *-<form>*	*Function*	Specify the format of the target program file. The presently supported formats are listed in Table 3 below. The format name *form* is case-insensitive.
	Default	Automatic detection of the file format.
-logues=call	*Function*	For an UBROF target program (compiled by the IAR compiler), specify that the prologue and epilogue "helper" routines be modelled as normal subprograms that are called from the subprograms that use them. See section 6.3.
	Default	The default is *-logues=integrate*, which see.
-logues=integrate	*Function*	For an UBROF target program (compiled by the IAR compiler), specify that the prologue and epilogue "helper" routines be modelled as integrated parts of the subprograms that use them, as if the option "integrate" were asserted for each prologue and epilogue routine. See section 6.3 and *-logues=call*.
	Default	This the default.
-mul	*Function*	The **mul** and **muls** instructions are modelled exactly as multiplications. Other multiplication instructions are modelled as giving an unknown result.
	Default	The default is *-no_mul*.
-no_mul	*Function*	All multiplication instructions are modelled as giving unknown results.
	Default	This is the default.
-no_switch_eval	*Function*	Disables partial evaluation of switch handler routines.
	Default	The default is *-switch_eval*, which see.

Option		*Meaning and default value*
-protocol=<name> or just *-<name>*	*Function*	Choose the calling protocol to be assumed for stack handling and parameter passing between subprograms, by giving the *name* of the protocol. The presently supported calling protocols are listed in Table 5 below. The protocol *name* is case-insensitive.
	Default	The target program file (or its format) may imply a default protocol. Otherwise there is no default and the protocol must be chosen with this option.
-switch_eval	*Function*	Enables partial evaluation of switch handler routines using the method described in reference [9].
	Default	This is the default.
-switch_offset=any	*Function*	Asserts that any offset added to the switch-table pointer within a switch handler may have any value, including a negative value.
	Default	The default is *-switch_offset=pos*, which see.
-switch_offset=pos	*Function*	Asserts that all offsets added to the switch-table pointer within a switch handler are non-negative.
	Default	This is the default.
-switch_steps=<N>	*Function*	Sets the maximum number *N* of flow-graph steps (instructions) for any subprogram that invokes a switch handler, when the switch handler is partially evaluated (*-switch_eval*). Analysis of the switch handler is aborted if the flow-graph reaches this size. This may happen if the partial evaluation of the switch handler is not precise enough to detect the end of the switch table.
	Default	The default is *-switch_steps=2000*.
-swstack=<D><P>	*Function*	Defines the auxiliary, software stack mechanism by means of the two one-character symbols *D* and *P* which define respectively the direction of growth and the stack pointer register. If *D* is '+' the stack grows upwards to higher addresses. If *D* is '–' the stack grows downwards to lower addresses. The letter *P* defines the stack pointer to be one of the three AVR pointer registers **X**, **Y**, or **Z** by the corresponding letter 'X', 'Y' or 'Z' or the lower-case equivalents. For example, *–swstack=–Y* defines a downwards-growing stack with register **Y** as the stack pointer. This is the most common form of software stack on the AVR.
	Default	There is no general default, but the chosen or implied calling protocol may imply a default software stack mechanism. See Table 5.

Program loading options

The two following tables describe the options that guide the process of reading the memory image and symbol tables of the target program to be analysed. Table 3 shows the possible values of the *-format* option and Table 4 describes some format-specific options.

Table 3: Supported Target Program File Formats

The *<form>* for *-format=<form>*	Format	Typical file-name suffix
COFF	Common Object File Format [7] from the ImageCraft ICCV7 compiler. No implied protocol. Use ICCV7 or ICCV7a with the *-protocol* option. Support for COFF symbols (debugging information) is rudimentary at present.	various
ELF	Executable and Linking Format from the GNU GCC compiler. Implies the GCC calling protocol.	.elf
UBROF	Universal Binary Relocatable Object Format from the IAR compiler. Implies the ICCA90 calling protocol.	.d90

Table 4: Options for Specific File Formats

Format	Option		Meaning and default value
COFF	(none)		
ELF	*-elf_sym*	*Function*	Makes Bound-T use the ELF symbol-table in addition to the STABS symbol table.
		Default	The ELF symbol-table is used only if the executable file has no STABS symbol-table.
UBROF	*-draw_classes*	*Function*	Draw the class hierarchy diagram, from the C++ class information in the UBROF file.
		Default	The diagram is not drawn.
	-draw_class_functions	*Function*	As *-draw_classes* but also shows names of function members in each class.
		Default	The diagram shows only the class name.
	-draw_class_members	*Function*	As *-draw_classes* but also shows names of data members.
		Default	The diagram shows only the class name.

Calling protocol options

Table 5 below lists the calling protocols that can be chosen with the option *-protocol=<name>*.

Table 5: Supported Calling Protocols

The *<name>* for *-protocol=<name>*	Calling protocol
GCC	The GNU C compiler calling protocol. See section 6.5. This protocol uses only the normal (**SP**) stack, no software stack. This protocol is well supported only for an 8-bit **SP**.

The *<name>* for *-protocol=<name>*	Calling protocol
IAR	Either of the two calling protocols used in the IAR C/C++ compiler. See section 6.3. Implied by default for UBROF executable files. Implies a software stack as in *–stack=–Y*.
ICCV7	The ImageCraft ICCV7 compiler calling protocol when the compiler option *-r20_23* is not used. See section 6.4. Implies a software stack as in *–stack=–Y*.
ICCV7a	The ImageCraft ICCV7 compiler calling protocol when the compiler option *-r20_23* is used. See section 6.4. Implies a software stack as in *–stack=–Y*.

AVR-specific -trace options

Table 6 below describes the AVR-specific items for the generic option *-trace,* to ask for certain additional outputs from Bound-T.

Table 6: AVR-Specific *-trace* Options

-trace item	Traced information
bref	Displays the arithmetic effect of each instruction as it is decoded and modeled (as for the generic option *-trace effect*) but puts each assignment on its own line, for a more readable listing.
classes	The class types (in the C++ sense) defined in the program. This information is available only for UBROF programs.
class_members	Class types and class members (in the C++ sense) defined in the program. This information is available only for UBROF programs.
finish	Finishing the arithmetic model to include composite cells – register pairs and pointers.
load	Program elements (segments, sections, symbols, ...) as they are loaded from the executable file.
logues	Routines classified as prologue or epilogue routines, as they are detected.
switches	Detection and analysis of switch handlers.
virtuals	Virtual function calls and possible callees. This information is available only for UBROF programs.

2.5 Supported AVR devices

Atmel produces many different processor chips – devices – with the AVR architecture. All these devices support most of the AVR instruction set, but have different sets of on-chip I/O peripherals and different amounts memory. Depending on the memory size, the code address (PC) may need two or three octets and the data address (AVR “pointer” registers) may need one, two, or three octets. The code-address size determines the number of cycles and the stack space needed for calls and returns (pushing and popping the PC). The data-address size determines how many octet registers combine to a “pointer” register and take part in auto-

increment or decrement. Both factors influence the operation of Bound-T, and therefore you must use the command-line option *-device name* to tell Bound-T which AVR device is to be used.

Table 7 lists the names of the AVR devices that Bound-T supports at the time of writing (use the *-help* option to get an up-to-date list). The names are case-insensitive. Future versions of Bound-T/AVR will have alternative options to define the relevant properties, such as PC size, for AVR devices that are not known to Bound-T by name.

Table 7: Supported AVR devices

The *<name>* for *-device=<name>*	The *<name>* for *-device=<name>*
AT90CAN128	*ATmega32*
AT90S1200	*ATmega32L*
AT90S8515	*ATmega64*
ATmega103	*ATmega64L*
ATmega103L	*ATmega644*
ATmega128	*ATtiny11*
ATmega128L	*ATtiny12*
ATmega163	*ATtiny13*
ATmega163L	

2.6 Choice of procedure calling protocol

The definition-analysis and (especially) arithmetic analysis of a subprogram depend on the calling protocol of the subprogram.

Bound-T chooses the calling protocol as follows:

- Command-line options *-protocol=<name>* or just *-<name>* (see Table 5).
- The calling protocol implied by the executable file (see Table 3).

Bound-T emits an error message if the executable file implies no calling protocol and no protocol is chosen with command-line options.

Bound-T emits a warning message if the executable file implies a calling protocol but a command-line option chooses a different protocol. The command-line option overrides the implied protocol from the executable file.

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 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,
- 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 AVR-specific aspects of the assertion language.

3.2 Naming items by address

Subprograms and code addresses

A subprogram can be named by giving its entry address in hexadecimal and in octet units. For example, the following assertion applies to the subprogram that is entered at the octet address A6E2 hex, corresponding to the instruction word address 5371 hex:

```
subprogram "a6e2"
   loop repeats 10 times; end loop;
end "a6e2";
```

The octet address must be an even number since it is the address of an instruction word. An odd address will be rejected with an error message.

Other code addresses (*eg*. for loop identification) are also given in octet units as hexadecimal numbers.

Code-address offsets

Some forms of assertions define code addresses by giving a *code offset* relative to a base address. For Bound-T/AVR 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 157B hexadecimal. The subprogram with the entry address 14D2 hexadecimal can then be identified in any of the following ways, among many others:

- Using the absolute address:

```
subprogram address "14D2"
```

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

```
subprogram "Rerun" offset "26"
```

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

```
subprogram "Abandon" offset "-A9"
```

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

Variables: registers and memory locations

Assertions can refer to program variables using machine-level names or addresses. For example, the assertion

```
variable address “p6” <= 102;
```

states that the 16-bit variable represented by the AVR register pair **r7**:**r6** has a value less or equal to 102.

The machine-level name of a variable consists of a *prefix* of one or more letters often followed by a *selector*. The selector can be a number or a mnemonic. The prefix defines the type of register or memory and the selector identifies the specific register or memory location. The table below shows the available prefixes and the corresponding selectors whether numeric or mnemonic and their meaning. The name is case-insensitive; for example the forms “r5” and “R5” are equivalent. The “Base” column shows the numeric base for numeric selectors.

Table 8: Naming Variables by Address

Prefix	Selector	Base	Meaning	Examples
r	0 .. 31	10	An 8-bit general register	**r0**, **r31**
p	An even number 0 .. 30	10	A 16-bit register pair formed of an even-numbered 8-bit register (selector, low octet) and the next 8-bit register (selector + 1, high octet).	**p0**, **p30**
p	**W**, **X**, **Y**, **Z**	-	**pW** is the same as **p24**. **pX** is the same as **p26**. **pY** is the same as **p28**. **pZ** is the same as **p30**.	**pX**
a	**X**, **Y**, **Z**	-	The pointer variables **X**, **Y** and **Z** as used in the AVR indirect-load and indirect-store instructions. See Table 9.	**aY**
RAMP	**X**, **Y**, **Z**, **D**	-	The AVR pointer-extension registers for the **X**, **Y** and **Z** pointers and for direct load and store (**RAMPD**).	**RAMPZ**
d	0 .. $2^{24} - 1$	16	A data memory octet.	**df8c**

dw	0 .. $2^{24} - 1$	16	A data memory word consisting of the octet at the given address (selector) and the following octet.	**dw34a0**
c	0 .. $2^{24} - 1$	16	A code memory octet.	**df8c**
cw	0 .. $2^{24} - 1$	16	A code memory word consisting of the octet at the given address (selector) and the following octet.	**cwff062**
io	0 .. 3F	16	An octet I/O register at the given I/O address (selector). Same as **d** with the selector increased by 32 = 20 hex.	**io0** = **d20** **io3f** = **d5f**

The meaning of the pointer variables “aX”, “aY” and “aZ” for data-memory access depends on the AVR device as shown in the table below. The significant factor is the memory size because it defines how many 8-bit registers are needed to form a memory address. In other words, in an AVR device with no more than $2^8 = 256$ octets of data memory the name "aX" is equivalent to the name "r26" and both identify the register **r26**. In devices with more than 2^8 but no more than 2^{16} octets of data memory "aX" is equivalent to "p26", the register pair **r26:r26**, while in devices with more than 2^{16} octets "aX" means the register triple **RAMPX:r27:r26**.

Table 9: Meaning of Pointer Variable

Variable	8-bit address	16-bit address	24-bit address
aX	**r26**	**r27:r26** = **p26**	**RAMPX:r27:r26**
aY	**r28**	**r29:r28** = **p28**	**RAMPY:r29:r28**
aZ	**r30**	**r31:r30** = **p30**	**RAMPZ:r31:r30**

For program-memory access the registers used in the address are defined by the instruction, not (directly) by the size of the program memory. The **lpm** (load program memory) instruction uses the 16-bit address formed by the register pair **r31:r30**, identified by the name "pZ", to read data from the program memory. This instruction can access the first 64 KiB of program memory. In contrast, the **elpm** (extended load program memory) uses the 24-bit address formed by the triple **RAMPZ:r31:r30** to read data from the program memory, up to 16 MiB in size. This triple is identified by the name "aZ". Naturally, AVR devices with no more than 64 KiB (32 kilo-words) of program memory tend not to implement the **elpm** instruction.

3.3 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 a branch instruction performs a branch or a call. The roles and their names are target-specific. The AVR version of Bound-T defines no assertable roles; it chooses the role of each instruction based on its own analysis of the instruction and its context.

3.4 Properties

The assertable properties for the AVR are listed and explained in the following table.

Table 10: Assertable Properties

Property name			Meaning, values and default value
virtual	*Function*		Controls the analysis of virtual function calls from a particular subprogram. Relevant only in subprogram scope (because applied during flow tracing).
	Values	0	Model virtual function calls as a set of alternative static calls, as with the option *-virtual static*.
		1	Model virtual function calls as dynamic (boundable) calls, as with the option *-virtual dynamic*.
		other	Undefined and reported as errors.
	Default		According to the setting of the command-line option *-virtual*.

4 THE AVR AND TIMING ANALYSIS

4.1 The AVR

An 8-bit RISC microcontroller

The AVR [4] is an 8-bit microcontroller core. It has a "Harvard" architecture (separate program and data memories) and a two-stage pipeline with separate fetch and execute cycles. Computational instructions use register and immediate operands and destinations; there are separate load and store instructions, as in RISC processors.

The size of the program and data addresses depends on the particular AVR device, according to the size of the program and data memories. A program address is 16 or 22 bits. A data address is 8, 16 or 24 bits.

Integer addition, subtraction and multiplication are supported in hardware but division is not. Integer operands are 8 or 16 bits long. The core AVR does not support hardware floating point operations.

Data memory is addressed by octet. Load and store instructions operate on 8-bit quantities. To load or store multi-octet values as many load or store instructions must be used. This means that there is no hardware-defined endianness in memory. The software (compiler) decides if multi-octet values are stored in little-endian or big-endian form. (However, the I/O area may contain some 16-bit quantities with a hardware-defined endianness, see below.)

Data memory cannot be bit-addressed (except for the I/O space, see below) but registers can be. Data octets must be brought into registers for bit operations.

Program memory is addressed by 16-bit word when fetching instructions. The program counter is a word address and thus increments by one for 16-bit instructions. However, data (constants) can be fetched from the program memory using octet addressing and one of the pointer registers (**Z**) about which more below.

Most instructions are 16 bits long. Some instructions have a second word and are thus 32 bits long.

Registers

There are 32 general 8-bit registers, **r0** through **r31**, but the instruction set is not entirely symmetric and some registers have special roles. Some instructions operate on register pairs where an even-numbered register holds the low octet of a 16-bit quantity and the next register holds the high octet. For example, the symbol **r31**:**r30** denotes the register pair formed of the registers **r30** (low octet) and **r31** (high octet). The eight highest-numbered registers form four pairs that have special roles and are called **W** = **r25**:**r24**, **X** = **r27**:**r26**, **Y** = **r29**:**r28**, **Z** = **r31**:**r30**. The **X**, **Y** and **Z** register-pairs can be used as data address registers (index registers) with optional auto-increment or auto-decrement. The **Z** register can be used as a code address register for indirect jumps and calls and to load constant data from the program memory with octet addressing (the **lpm** and **elpm** instructions).

The 32 general registers are also mapped in the start of the data address space at addresses 0 through 31. This means that registers can also be accessed indirectly.

In addition to the general registers, there is a Program Counter (**PC**) register, a Stack Pointer (**SP**) register and a Status Register (**SREG**) that contains the condition flags and interrupt masks.

The Program Counter **PC** points to the next instruction in the program memory. Depending on the AVR model (size of code memory) the **PC** is either 16 bits or 22 bits wide. This influences the timing of some instructions and the stack space required for return addresses.

The Stack Pointer **SP** is used in call and return instructions to push and pop the return address. There are also push and pop instructions for storing data in the stack. The stack grows downwards; a **push** post-decrements **SP**. In some small AVR chips the call/return stack is not located in the general RAM but in a small, special memory with space for only a handful of return addresses. In such chips the call/return stack pointer is invisible to the program and there is no **SP** register and no push/pop instructions for data storage.

In addition to the processor stack many compilers for the AVR define and use a *software stack*. The compilers use the processor stack for return addresses and the software stack for parameter data and local variable data. Space for the software stack is allocated in the data memory and the software stack pointer is usually the **Y** register pair.

Address extension registers

AVR devices with more than 64 kilo-octets of data memory have dedicated registers that extend the **X**, **Y** and **Z** register-pairs with further high-order address bits. The **RAMPX** register extends **X**, the **RAMPY** register extends **Y** and the **RAMPZ** register extends **Z**.

For indexed jumps and calls the **Z** register-pair is extended by a different extension register called **EIND**.

When an instruction uses an immediate 16-bit address to access data memory, it can be extended to a 24-bit address by an extension register called **RAMPD**.

Condition flags

The status register **SREG** contains the conventional condition flags (**Z** = zero, **N** = negative, **C** = carry, **V** = overflow) and a general flag (**T**). The **T** flag can be loaded and stored from or to a specific bit in a specific general register. There is also a “signed comparison” flag **S**, which is defined to be the exclusive-or of **N** and **V**, and a half-carry flag **H** for use in Binary Coded Decimal arithmetic. We will ignore the **S** flag because it can be computed from **N** and **V**. We will ignore the **H** flag because it is unlikely to be relevant for our purposes (Bound-T does not model the **H** flag at all).

I/O registers

A part of the data address space is called the I/O address space and is accessed with dedicated **in** and **out** instructions or with dedicated bit-setting and bit-testing instructions. The I/O address space contains the peripheral control and data registers (which, of course, depend on the processor model). The I/O address space also provides access to some general registers: the **SREG**, the low and high octets of **SP** (**SPL**, **SPH**) and the address extension registers **RAMPX**, **RAMPY**, **RAMPZ**, **RAMPD** and **EIND**. For 16-bit quantities like the **SP** the I/O area uses little-endian order (**SPL** comes before **SPH**).

Since the I/O adddress space is a part (range) of the data address space, I/O registers can also be accessed using normal data load and store instructions.

Some AVR devices have more I/O registers than can be addressed by **in** and **out** instructions, so load and store instructions must be used for the remainder. This area is called the extended I/O space. The real data memory space (RAM space) starts after the extended I/O space.

Data memory, endianness

The AVR data memory is addressed by octet. The general and I/O registers are embedded in the data memory address space. All memory load and store operations work on octets. This means that there is no hardware-defined endianness: when a multi-octet quantity such as a 16-bit integer is stored in memory the order of the octets is defined by the software. The four register pairs combine the registers **r24** .. **r31** in little-endian order, which suggests that

software should use little-endian order in general. Moreover, when the **SP** is 16 bits its two 8-bit parts **SPL** and **SPH** are also mapped as I/O registers in little-endian order. On the other hand, the **call** instruction stores the return address on the stack in big-endian order.

Memory map

The following table shows an overview of the data memory map and how the general registers and I/O registers can be accessed with data memory addresses.

Table 11: Data Memory Map

Data address		I/O address		Content
hex	dec	hex	dec	
00	0			**r0**
01	1			**r1**
...	...			**r2** – **r25**
1A	26			**r26** = low octet of **X**
1B	27			**r27** = high octet of **X**
1C	28			**r28** = low octet of **Y**
1D	29			**r29** = high octet of **Y**
1E	30			**r30** = low octet of **Z**
1F	31			**r31** = high octet of **Z**
20	32	00	0	First I/O register. Meaning depends on the AVR device.
21	33	01	1	Second I/O register, ditto.
...	...	...	...	...
5F	95	3F	63	Last I/O-addressable I/O register, ditto.
60	96			First location that is not I/O addressable. For some AVR devices this is the first true memory octet (storage cell); for others this is the first extended I/O space address.
61	97			Second location, ditto.
...	...			...
FFFF	65 535			Last memory location addressable with 16 bits, that is, without using the extension registers (**RAMP** registers).
10000	65 536			Memory over 64 kilo-octets, if the AVR device has such. Accesses must use the extension registers (**RAMP** registers).
...	...			...

4.2 Static execution-time analysis on the AVR

The AVR architecture is very regular and quite fitting for static WCET analysis by Bound-T. Instruction timing usually depends only on the control-flow and is independent of the data being processed.

When a branch occurs, the AVR reloads the instruction pipeline before continuing. This means that there are no “delayed” branches, which simplies control-flow analysis.

The following architectural features can lead to approximate (over-estimated) execution times for the concerned instructions:

- Memory wait states that vary in number depending on the address, because some addresses map to on-chip, internal memory and others slower off-chip, external memory.
- Flash-memory access and buffering delays, when the AVR device uses some kind of buffering, caching, or prefetching for the flash memory.

5 SUPPORTED AVR FEATURES

5.1 Overview

This section specifies which AVR instructions, registers and status flags are supported and modelled by Bound-T. We will first describe the extent of support in general terms, with exceptions listed later. Note that in addition to the specific limitations concerning the AVR, Bound-T also has generic limitations as described in the Bound-T Reference Manual [1]. For reference, these are briefly listed in section 5.2.

General support level

In general, when Bound-T is analysing a target program for the AVR, it can decode and correctly time all instructions, with minor approximations except for coprocessor instructions.

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 standards listed in chapter 6. 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 integer additions and subtractions for 8-bit and 16-bit integers, but not for wider integers, for example not for 32-bit integers. Bound-T correctly detects when this 8/16-bit integer computation is overridden by other computations, such as multiplications or wider integer computations. Note that there are generic limitations on the analysis of pointers to variables (aliasing).

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, the Bound-T user should not meet with any AVR-specific constraints for 8-bit and 16-bit integers but may be disappointed by the lack of analysis of wider integers, for example 32-bit integers.

5.2 Reminder of generic limitations

To help the reader understand which limitations are specific to the AVR architecture, the following compact list of the generic limitations of Bound-T is presented.

Table 12: Generic Limitations of Bound-T

Generic Limitation	Remarks for AVR target
Understands only addition, subtraction and multiplication by constants, in loop-counter computations.	No implications specific to the AVR.
Assumes that loop-counter computations never suffer overflow.	No implications specific to the AVR.
Can bound only counter-based loops.	No implications specific to the AVR.
May not resolve aliasing in dynamic memory addressing.	Analysis may be incorrect if the unresolved dynamic reference writes to the general registers **r0** .. **r31** (decimal addresses 0 .. 31) or to special registers such as **SREG** or **SP** in the I/O area (decimal addresses 32 .. 95).
May ascribe the wrong sign to an immediate (literal) constant operand.	No implications specific to the AVR.

5.3 Main assumptions

Bound-T for the AVR makes the following AVR-specific assumptions about the target program under analysis:

- The registers **r0 .. r31**, the status register **SREG**, the stack pointer register **SP**, and the address-extension registers **RAMPX**, **RAMPY**, **RAMPZ**, **RAMPD**, and **EIND** are not changed by indirect access (access via pointers).
- The program memory is read-only. If the program reads data from the program memory, using an **lpm** or **elpm** instruction, and Bound-T can resolve the address that is read, and the executable file under analysis statically defines a value for address, this value is returned by the **lpm** or **elpm**.
- The **ret** and **reti** instructions always perform a return from the current subprogram (interrupt handler, for **reti**). The use of **ret** as a dynamic branch or call, to whatever address is pushed on the stack before the **ret**, is not now supported.
- The choice of cross-compiler and/or calling protocol may imply further assumptions.

5.4 Instructions and computations

Bound-T for the AVR models the main computational effect of most AVR instructions accurately, within the generic limitations of Bound-T and within the current AVR-specific limitation to 8-bit and 16-bit computations. This section describes the computational effects that are modelled approximately or not at all. However, note that some generic analyses in Bound-T may introduce generic approximations. For example, the loop-bounds analysis based on Presburger Arithmetic assumes that loop-counter computations do not overflow.

Registers and memory

Most registers and memory locations in the AVR are modelled. The following are modelled in limited ways:

- The absolute value of the **SP** register is generally opaque; only the changes in **SP** are modelled. The same holds when a pointer register is used as the stack pointer for a compiler-specific software stack.
- All memory locations, except I/O registers, are currently assumed to have ordinary non-volatile memory semantics, that is, reading the location returns the last-written value.
- Only the well-known I/O registers **SPL**, **SPH**, **RAMPX**, **RAMPY**, **RAMPZ**, **RAMPD**, and **EIND** are modelled as non-volatile storage, for which an **in** instruction returns the value written by the last **out** instruction. For all other I/O registers the result of **in** is opaque (an unknown value) and **out** is assumed to have no effect.
- Status register **SREG** flags **V**, **H**, **I** are not modelled. Their values are considered unknown.

Future versions of Bound-T will provide means to define which memory locations and I/O registers are “volatile” and which are not. Even when a memory location or register is physically non-volatile, for the analysis of a single thread in a multi-thread system it may appear to be volatile if its value is changed at unpredictable times by other threads running concurrently.

Instructions with unknown result

The **swap** (swap nibbles) and **ror** (rotate right through carry) instruction are currently given an unknown computational result in the affected register.

The instructions that change a single bit in an 8-bit octet are modelled as setting the whole octet to an unknown value. These instructions are **bld**, **cbi**, **sbi**.

Under default options, all multiplication instructions are given an unknown computational result. Under the option *-mul*, the instructions **mul** (multiply two unsigned numbers) and **muls** (multiply two signed numbers) are modelled as such multiplications, but the **mulsu** (multiply signed with unsigned) instruction, and all fractional multiplication instructions (**fmul**, **fmuls**, **fmulsu**), are still given unknown computational results.

Since the program memory is assumed to be read-only, the **spm** (store program memory) instruction is assumed to have no effect on any computation that is important for the analysis. Thus is it modelled as a no-operation instruction (with a warning).

Instructions with unknown timing

The **break** and **sleep** instructions have a non-deterministic effect on the real execution time. If they occur in code subject to timing analysis, Bound-T emits a warning.

The **spm** (store program memory) instruction writes to flash memory and has a variable execution time. Bound-T assumes only one cycle for the execution time of this instruction, and emits a warning for every **spm**.

Stack Pointer SP

The value of **SP** is tracked mainly relative to its value on entry to the subprogram under analysis (the local stack height in the subprogram).

For AVR devices with a 16-bit SP register, if the program changes the value of **SP** by **out** instructions that separately change the low and/or high octets **SPL** and **SPH**, Bound-T may not be able to compute the actual change in **SP** and the resulting stack usage. This problem currently happens for *gcc* and subprograms with large local variables.

5.5 Computations with 16-bit numbers

The problem

The main problem in modelling the arithmetic computations in AVR programs is the management of 8-bit registers *vs* 16-bit register pairs. In this section we explain the problem and (briefly) what Bound-T does about it and how.

Most AVR instructions operate on 8-bit operands (octets, bytes). The few instructions that operate on 16-bit operands (register pairs) are **adiw** (add immediate word) and **subiw** (subtract immediate word) in which the other operand is an immediate constant of limited magnitude, and **movw** (move word) which is just a register-to-register copy. The auto-increment and auto-decrement options in the indirect load and store instructions also operate on 16-bit (or even 24-bit) address values in the (possibly extended) **X**, **Y** and **Z** registers, but there it ends.

Therefore, most arithmetic on 16-bit or larger numbers in an AVR program must be implemented by *chains* of 8-bit operations such as **add** followed by **adc** (add with carry). If we did not detect and model such operation chains we would be unable to find loop-bounds automatically for loops with counters larger than 8 bits.

The detection and modelling of operation chains in Bound-T/AVR is currently implemented in AVR-specific ways. In the future, it will be implemented by generic, processor-independent methods.

Register pairs for 16-bit values

In principle a 16-bit value could be held in any two 8-bit registers, for example with the less significant octet (also called the low octet) in **r2** and the more significant (high) octet in **r5**. However, AVR compilers tend to follow the example set by the AVR pointer registers **X**, **Y**, **Z** which are composed of register pairs with adjacent numbers, with the low octet in a register with an even number and the high octet in the next (odd-numbered) register. For example, the **X** pointer consists of the register pair **r27**:**r26**, using the notation (high octet):(low octet).

Following this example, AVR compilers seem to keep all 16-bit values in even:odd register pairs, such as **r1**:**r0**. Bound-T detects and models 16-bit computations only under this condition, that register operands are odd:even register pairs.

Memory octet pairs for 16-bit values

As for registers, 16-bit values could be stored in memory using any pair of octets, but we assume that compilers will always use adjacent memory octets to store the low and high octets of a 16-bit value, and will use the same endianness (octet order) for all 16-bit values. However, different compilers may use different endianness, so the endianness that Bound-T assumes can be set by the command-line option *-endian*.

Bound-T thus detects and models 16-bit accesses to memory and I/O register only when the memory locations involved are adjacent and in the right endianness order.

Chaining 8-bit operations into 16-bit operations

Bound-T for the AVR detects and models some pairs (two-step chains) of 8-bit operations that implement 16-bit computations. When the two instructions are consecutive, such as an **add** immediately followed by an **adc**, the chain is detected and modelled on the fly as instructions are decoded and entered in the flow-graph. Non-consecutive chainable instructions are detected in a later phase based on data-flow analysis of the whole flow-graph.

The operations that can be chained in this way are loading values from memory, storing values into memory, loading a literal value, moving values between registers, addition, subtraction and comparison. Shifts and rotations are not currently chained. Table 13 below shows the chainable instruction pairs, the conditions under which a given pair is chained, and the chained (16-bit) effect. In general, two 8-bit operations are chained when the 8-bit destination registers (which are also the first 8-bit operands in each operation) form an *odd*:*even* register pair; this pair becomes the 16-bit destination register and the first 16-bit operand in the chained operation. It is not required that the second 8-bit operands in the 8-bit operations should form such a pair, but if they do then this pair becomes the second 16-bit operand in the chained operation.

Table 13: Chainable Instruction Pairs

Instruction		Chained effect	Chaining condition
first	**second**		
add	**adc**	16-bit addition	The destination registers form an odd:even pair, in the order second:first, and the first instruction sets (defines) the carry flag for the second instruction.
sub/subi	**sbc/sbci**	16-bit subtraction	
cp/cpi	**cpc**	16-bit comparison	
eor	**eor**	The register pair is set to zero.	The destination registers form an odd:even pair in either order.

Instruction first	Instruction second	Chained effect	Chaining condition
mov	**eor**	16-bit comparison to zero.	The **mov** and **eor** have the same destination register and the source registers form an odd:even pair in either order.
ldi	**ldi**	The register pair is set to the combined 16-bit value.	The destination registers form an odd:even pair in either order.
lds	**lds**	The register pair is loaded with the 16-bit value of the memory word.	The source octets (memory addresses) form a 16-bit word with the right endianness (option *-endian*), and the destination registers form an odd:even pair in the corresponding order.
sts	**sts**	The 16-bit value of the register pair is stored in the memory word.	The destination octets (memory addresses) form a 16-bit word with the right endianness (option *-endian*), and the source registers form an odd:even pair in the corresponding order.
ld/ldd	**ld/ldd**	The register pair is loaded with the 16-bit value of the memory word, addressed by pointer + offset.	Both instructions use the same pointer register; the destination registers form an odd:even pair; the offsets differ by 1; and the order of the offsets gives the right endianness (option *-endian*) in the memory word.
st/std	**st/std**	The 16-bit value of the register pair is stored in the memory word, addressed by pointer + offset.	Both instructions use the same pointer register; the source registers form an even:odd pair; the offsets differ by 1; and the order of the offsets gives the right endianness (option *-endian*) in the memory word.
push	**push**	The 16-bit value of the register pair is pushed onto the stack.	The source registers form an odd:even pair and the order in which they are pushed gives the right endianness (option *-endian*) in the memory word in the stack. For example, for little-endian order the odd (high) octet must be pushed before the even (low) octet (remember that the stack grows downwards).
pop	**pop**	The 16-bit value on top of the stack is popped into the register pair.	The destination registers form an odd:even pair and the order in which they are popped matches the right endianness (option *-endian*) in the memory word in the stack. For example, for little-endian order the even (low) octet must be popped before the odd (high) octet.
in	**in**	The register pair is loaded with the 16-bit value read from the I/O register pair ("I/O word").	The source octets (I/O addresses) form a 16-bit word with the right endianness (option *-endian*), and the destination registers form an odd:even pair in the corresponding order.
out	**out**	The 16-bit value of the register pair is written to the I/O register pair ("I/O word").	The destination octets (I/O addresses) form a 16-bit word with the right endianness (option *-endian*), and the source registers form an odd:even pair in the corresponding order.

The **cpi** (compare with immediate value) instruction is not chained because there is no "compare with carry" form and the processor does not chain the "equal" result (the **Z** flag) between successive **cpi** instructions.

5.6 Control-transfer instructions

The first and critical phase in Bound-T's analysis is to construct the control-flow graphs and the call graph of all the subprograms to be analyzed. This requires a decoding and modelling of all control-transfer instructions. Most AVR control-transfer instructions, such as **jmp** and **call** and the conditional branch instructions like **breq**, statically define the target address and pose no problem. In contrast, the *indirect* control-transfer instructions **ijmp** and **icall** and their extended variants **eijmp** and **eicall** use a dynamically computed target address which may or may not be resolved by Bound-T's analysis. The return instruction **ret** also uses a dynamically defined target address, in this case popped from the stack.

Indirect jumps

The **ijmp** instruction is modelled properly as a jump to the address defined by the **Z** register pair. Bound-T analyses this instruction as a possible jump to a table of further jumps, a code idiom sometimes generated for switch-case statements.

The **eijmp** instruction is not supported because it relies on 24-bit computation of the extended pointer **EIND**:**Z**. An error message is given for this instruction.

Indirect calls

An **icall** or **eicall** instruction is usually modelled as a dynamic call that cannot be resolved by analysis. Thus it must be resolved by an assertion that lists the possible callees.

However, if the **icall** or **eicall** represents a virtual-function call in a C++ program compiled by the IAR compiler, the possible callees can be found from the UBROF file because the UBROF debugging information contains the C++ class structure which identifies all the subclasses and their actual implementations of the virtual function. The model used then depends on the setting of the generic command-line option *-virtual*:

- Under *-virtual static*, Bound-T models the **icall**/**eicall** as a non-deterministic choice between static calls to each of the implementations of the virtual function, as defined in the UBROF file. Bound-T does not try to reduce the set of callees by analysis, for example by analysing the actual class of the "this" object.
- Under *-virtual dynamic*, Bound-T models the **icall**/**eicall** as a dynamic call that cannot be resolved by analysis. An assertion is then necessary to resolve the call.

The latter case (*-virtual dynamic*) lets the user decide which subclasses can really occur at this place in the program, by listing only the corresponding subset of possible callees.

Return instructions

At present, all **ret** and **reti** instructions are modelled as a return from the current subprogram. They cannot be modelled as other kinds of dynamic transfer of control.

5.7 Stack-usage analysis

Processor stack (SP)

Bound-T analyzes stack usage on the normal **SP** stack by analyzing how the analyzed code changes the **SP** register. The absolute value of the **SP** register is seldom visible to the analysis, and is not an objective of the analysis.

The instructions that change the SP as a whole are the call instructions (**call**, **icall**, **eicall**); the return instructions (**ret**, **reti**); and the **push** and **pop** instructions. All these instructions change the SP by constant amounts and pose no problems for the stack-usage analysis.

However, in most AVR devices the low and high octets of the **SP** register, **SPL** and **SPH**, are also accessible as I/O registers and can thus be changed by **out** instructions. When a program needs to change the **SP** by a largish amount, say decrease it by 713 to allocate 713 octets of local stack space for the current subprogram, the program will read the **SP** by two **in** instructions from **SPH**:**SPL** into a register pair, use a pair of **sub** (or **subi**) and **sbc** (or **sbci**) instructions to subtract 713 from the register pair, and write the result back to the **SP** by two **out** instructions. In its present form, Bound-T/AVR does not always model such computations (especially if optimized in some way) well enough to deduce the resulting overall change in the value of **SP**. This hampers stack-usage analysis for *gcc*-compiled programs, because *gcc* uses the **SP** stack for local variables.

Software stacks

Because the AVR provides no direct SP-relative addressing, many AVR cross-compilers define a second, "software" or "data" stack for local variables and parameters, and use the **SP** stack only for return addresses. Bound-T/AVR supports the use of any one of the three pointer registers, **X**, **Y**, **Z** as the stack pointer. The stack can grow upwards or downwards. The stack pointer and the growth direction are set by the command-line option -swstack, or by default for the chosen compiler, as described in section 2.4.

As for the **SP** stack, the analysis of stack usage in a software stack focuses on the changes in the stack pointer, not the absolute value of the pointer. The code that manages software stacks tends to be rather easier for Bound-T to analyze, than the corresponding code for the **SP** stack, and so stack-usage analysis currently works better for compilers that use a software stack (in addition to the **SP** stack).

6 SUPPORTED COMPILERS

6.1 Introduction

Bound-T analyses the binary code following the definition of the AVR instruction set. Ideally this should make it possible to analyse any code, produced by any compiler or by manual coding in assembly language. In practice, the analysis methods in Bound-T make certain assumptions on how the code behaves which means that some forms of code cannot be analysed or are difficult to analyse. The assumptions concern the following aspects:

- Procedure calling conventions and parameter-passing conventions.
- Stack usage conventions.
- Use of dynamic (indirect) jumps and calls, in particular for switch-case statements or virtual function calls.

This chapter explains the assumptions that Bound-T for the AVR makes on these aspects of the code to be analysed and how these assumptions are satisfied for the following compilers:

- The IAR C/EC++ compiler for AVR [8]
- The ImageCraft ICC V7 C compiler for AVR [6]
- The GNU C compiler for AVR [5].

The information in this chapter is to some extent preliminary and may be incomplete or describe foreseen rather than currently implemented functionality in Bound-T for the AVR.

6.2 Procedure calls in the AVR

In this chapter, we discuss how AVR 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. For the AVR architecture this is a little more complex than usual, for reasons that will be explained below. This means that you should know a little about the procedure calling standards and should know or choose the standard used in your program before using Bound-T.

Calling protocols

The AVR instruction set has dedicated instructions for calling subprograms (**call**, **rcall**, **icall**, **eicall**) and for returning from subprograms (**ret**, **reti**). All other aspects of subprogram calling, such as the passing of parameters, the saving and restoring of registers, and the use of the stack, are defined by *software* rules. Such rules are usually called a *procedure calling standard* or *calling convention* or *calling protocol*.

This flexibility (or weak standardisation) means that Bound-T must be told which calling protocol is used in the target program to be analysed. Moreover, Bound-T understands and supports only a limited set of calling protocols, as follows:

- Both protocols that can be used in the IAR C/EC++ compiler for the AVR [8].
- The protocol used in the ImageCraft ICC V7 compiler for the AVR [6], in two forms depending on the ICC V7 compiler option *-r20_23*.
- The protocol used in the GNU C compiler for the AVR [5], although support for this protocol is quite limited at present.

In the remaining subsections 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 rules need not be followed.

The supported protocols have several common features:

- The general registers are divided into two groups, the *volatile* registers that can be changed by any subprogram, and the *preserved* registers that are assumed to preserve their value across any subprogram call (thus the callee must save and restore these registers if it uses them).
- The return address is always passed in the hardware stack using the **SP** register as defined for the AVR instructions **call**, **ret** and so on.
- Parameters may be passed in certain registers or in a stack.
- Function results are returned in certain registers, or indirectly through a passed pointer to a result area allocated by the caller.
- The stack for parameters (and local variables) may be the hardware stack or a *software stack* which is a RAM area allocated by the compiler and accessed through a dedicated pointer register, usually the **Y** register.

Auxiliary software stacks

Several compilers use the AVR hardware stack only for return addresses and define an additional, auxiliary software-managed stack for passing parameters and holding local variables. Bound-T can analyse six kinds of software stack, as follows:

- The stack pointer can be one of the three index registers **X**, **Y** or **Z**.
- The stack can grow up, towards higher memory addresses, or down, towards lower memory addresses.

The kind of software stack used in a given target program can be implied by the choice of calling protocol or it can be defined by the command-line option *-swstack=<D><P>* where *D* defines the direction ('+' for up, ' –' for down) and *P* is 'X', 'Y' or 'Z' (or the lower-case equivalents) to define the stack pointer register. The most common form is *-swstack=–Y*, a stack that grows downwards and uses register **Y** as the stack pointer.

The definition of the software stack determines the instructions that Bound-T models as stack pushes, stack pops, or accesses to stack-allocated data (parameters or local variables). For example, in a "–Y" stack, a push is an **st** instruction of the form **st -Y,rn** (decrement **Y** and then store register **rn** in memory at address **Y**) and a pop is an **ld** instruction of the form **ld rn,Y+** (load register **rn** from memory at address **Y** and then increment **Y**).

Prologue and epilogue routines

Subprograms using these calling protocols often start by storing several callee-save registers on the stack and end by restoring the same registers from the stack. The compilers therefore usually provide several library routines, here called *prologue* and *epilogue* routines, that can be called to store and restore registers in this way. The compiler often inserts calls to these routines in subprograms that use callee-save registers.

The prologue routines push callee-save registers on the stack and the epilogue routines pop them from the stack. Thus, these routines do *not* themselves follow the calling protocol (instead, they *implement* parts of this protocol). It would be contradictory for Bound-T to model these routines as ordinary subprograms which are expected to following the protocol, for example to preserve the height of the stack. Therefore, Bound-T tries to analyse all

prologue and epilogue routines as “integrated” routines. This means that the instructions in the prologue or epilogue routine are modelled as integrated parts (steps) of the flow-graph of a subprogram that calls the prologue or epilogue.

Consequently, the prologue and epilogue routines are not given flow-graphs of their own, all their execution time and stack usage is included in the execution time and stack usage of the subprograms that call them, and they do not appear in the call-graph of the program.

Bound-T can use two methods to classify a given subprogram as a normal subprogram or a prologue or epilogue routine:

- the symbolic name (identifier) of the subprogram, or
- the instructions in the subprogram.

Which method(s) are used can depend on the chosen (or implied) calling protocol and on Bound-T command-line options.

When Bound-T uses the instructions in a subprogram to detect prologues and epilogues it generally uses the following definition:

- A routine that consists entirely of a sequence instructions that push different registers on the stack, followed by a **ret** instruction, is a *prologue* routine.
- A routine that consists entirely of a sequence instructions that pop different registers from the stack, followed by a **ret** instruction, is an *epilogue* routine.

A compiler often implements a call to an epilogue routine as a jump instruction, instead of a call instruction, because the call is a “tail call”. For such epilogue “calls” Bound-T always integrates the epilogue routine in the caller's flow-graph because Bound-T does not recognise the jump as a call.

Note that integrated decoding can be requested for specific subprograms by means of an “integrate” assertion as explained in the Bound-T Assertion Language Manual [3].

6.3 The IAR C/EC++ compiler

General

Bound-T can TBA.

The IAR C/EC++ compiler provides a choice of two calling protocols, called “calling conventions” in [8]:

- the original or “version 1” protocol, also called ICCA90 and activated with the command-line option *--version1_calls*,
- the new calling protocol, which is the default.

Bound-T can analyse programs that use either protocol. Both protocols use the same sets of volatile and preserved registers and use the processor stack and the auxiliary software stack in the same general way; the protocols differ only in the algorithm that chooses which parameters to pass in registers. Bound-T does not depend on this algorithm and thus both IAR protocols are equivalent to Bound-T.

The IAR Calling Protocols

Introduction

The two IAR calling protocols have many common features that are described in this section.

Stacks

Both protocols use the processor stack (**SP** stack) for return addresses, but not for data such as local variables or parameters; data are placed on the compiler-defined software stack with the **Y** register as the stack pointer.

Data on the software stack are stored in little-endian order: the least significant octet has a small address than the more significant octets.

Parameter passing

When parameters are passed on the stack (that is, the compiler-defined software stack with the **Y** register as the stack pointer) they are pushed on the stack in some order. The choice of parameters to be passed in the stack and the order of pushing can differ between the two IAR protocols.

For multi-octet data the octets are pushed in decreasing significance order which results in little-endian storage order.

Return from subprogram

The callee is responsible for popping the stacked parameters (that were pushed by the caller). Thus, the overall effect of a subprogram usually decreases the height of the software stack (more pops than pushes).

The callee returns with the normal **ret** instruction. Interrupt subprograms return with the **reti** instruction. These instructions pop the return address (and the **SREG**, for **reti**) from the processor stack, so the overall effect of a subprogram usually decreases the height of the processor stack.

IAR Prologue and Epilogue Routines

The IAR compiler uses several prologue and epilogue routines that push callee-save registers on the **Y**-stack or pop them from the **Y**-stack. As discussed above (section 6.2) these routines should be decoded as integrated parts of the calling subprograms.

Bound-T uses the symbolic name (identifier) of a routine to classify it as a normal subprogram or a prologue or epilogue. When a program is loaded from an UBROF file, any routine with a name that starts with the string ?PROLOGUE is considered a prologue routine, and any routine with a name that starts with the string ?EPILOGUE is considered an epilogue routine.

This default behaviour (integrating all prologue and epilogue calls) can be disabled with the command-line option *-logues=call*. This option turns off the detection of prologue and epilogue routines which means that calls to these routines are modelled in the normal way (as references to the callee's flow-graph) unless integrated decoding is specifically requested for specific routines by means of an “integrate” assertion as explained in the User Manual [].

However, the IAR compiler often implements a call to an epilogue routine as a jump instruction, instead of a call instruction, because the call is a “tail call”. For such epilogue “calls” Bound-T always integrates the epilogue routine in the caller's flow-graph, even under the option *-logues=call*.

IAR Switch-Case Statements

The IAR compiler can generate several different types of code for switch-case statements, depending on the form of the statement and on compilation options. Table 14 below lists these forms, shows the value to be used in the IAR option *--force_switch_type* to make the compiler generate this form, and explains if and how Bound-T can analyse each form.

Table 14: IAR Options for Switch-Case Statements

Form of code	*--force_switch_type*	Analysis in Bound-T
Library call with switch table	0	Analysed by partially evaluating the library routine (the switch handler) with respect to the (constant) switch table. See [9]. Depends on the Bound-T option *-switch_eval*, see Table 2.
Inline code with switch table	1	Under investigation.
Inline compare/jump logic	2	Analysed as part of the normal control-flow analysis. No special action is necessary.

The switch-table form (*--force_switch_type 0*) may reduce the code size if the program has many switch-case statements, but the inline compare/jump logic (*--force_switch_type 2*) is typically much faster in execution.

6.4 The ImageCraft ICCV7 compiler

General

Bound-T can analyse programs created with the ImageCraft ICCV7 compiler [6], provided that the linker is set to generate the executable program as a COFF file [7].

The ICCV7 Calling Protocol

The caller pushes stacked parameters on the software stack (**Y** is stack pointer).

Parameter and result passing

Scalar parameters are passed in registers **r16** - **r19** allocating them in order of increasing number but using two registers also for 8-bit parameters (the high-octet, odd-numbered register is then not used).

If **r16** and **r17** are used to pass the first parameter and the second parameter is a 32-bit type (long or float) the lower half of the second parameter is passed in **r18** and **r19** and the upper half is passed on the software stack. However, this is not relevant to Bound-T because Bound-T models neither long nor float values.

The remaining parameters are passed on the software stack. The **Y** register is the software-stack pointer.

Structure parameters passed by value are always passed on the software stack, never in registers. Structure parameter passed by reference are of course (scalar) pointers and may be passed in registers or on the software stack.

A scalar result of a function is returned in registers **r16** .. **r19** in the same way. For functions that return structures the caller passes a pointer to storage allocated in the caller and the function stores its results through this pointer, as if the structure were passed by reference.

Preserved registers

The ICCV7 compiler assumes that the registers **r10** - **r15**, **r20** - **r23** and **r28** - **29** (**Y**) are preserved across a call. (To be precise, the manual [6] says that assembly language subprograms must preserve these registers; it is unclear if this applies to C functions in globally optimized C programs.)

The compiler option *-r20_r23* makes the ICCV7 compiler leave registers **r20** - **r23** unused, which means that the application programmer is free to use these registers in any way, and they need not be preserved across calls. Bound-T uses the name "ICCV7a" for this variation of the ICCV7 protocol (see Table 5, Supported Calling Protocols).

Volatile registers

The **SREG** and the general registers that are not preserved, thus **r0** - **r9**, **r16** - **r19**, **r24** - **r27** and **r30** - **r31**, are considered volatile and can be changed by any subprogram call. If option *-r20_23* is used then Bound-T considers registers **r20** - **r23** to be volatile too.

Return from subprogram

The callee pops the stacked parameters from the software stack. Note that this means that register **Y** is not preserved across all calls (TBC with ImageCraft).

ICCV7 prologue and epilogue routines

The ICCV7 compiler uses several prologue and epilogue routines to push or pop callee-save registers and parameter registers onto or off the software stack, with **Y** as the stack pointer. As discussed above (section 6.2) these routines should be decoded as integrated parts of the calling subprograms.

The symbolic debug information in a COFF file from ICCV7 does not give any symbolic names (identifiers) to the prologue and epilogue routines. Therefore Bound-T inspects the routine contents (instructions) to detect prologues and epilogues as described in section 6.2.

ICCV7 switch-case statements

Under investigation.

6.5 The GNU GCC compiler

General

The GNU *gcc* compiler is widely used for AVR programming. Although *gcc* is commonly considered to be more adapted to 32-bit processors the AVR port of *gcc* seems to be quite good, even compared to more specialized compilers. Unfortunately, at present Bound-T/AVR supports *gcc* poorly, because *gcc* generates code that is rather different from that generated by other AVR compilers — one major difference being that *gcc* uses the **SP** stack for parameters and local variables, where other compilers use software-defined stacks.

GCC Calling Protocol

Introduction

This section explains the procedure calling protocol (also called the "calling standard") used by the *gcc* compiler for the AVR. The description is taken from the AVR-Libc documentation [5]. The AVR *gcc* calling protocol has the following features:

- The hardware stack (**SP** register) is used for parameters and local variables as well as for return addresses. There is no auxiliary software-defined stack.
- Prologue and epilogue sequences can be generated in-line (default) or as shared routines that are invoked from the application subprograms (compiler option *-mcall-prologues*).
- For in-line prologues, the option *-mtiny-stack* makes the compiler generate push/pop code that changes only the low octet (**SPL**) of the stack pointer (see section 6.5).

- Register **r0** is a general scratch register.
- Register **r1** is always zero in C code. It can be nonzero only in subprograms written in assembly language, and must be restored to zero when such a subprogram returns to C code.

Endianness

Word data (16 bits) are stored in memory in little-endian order: first the low octet, then the high octet.

Register usage

Table 15 below describes how *gcc* uses the AVR registers, in particular which registers are invariant over a call (callee-save).

Table 15: GCC Register Usage

Register	Usage	In a call
r0	General scratch register.	Can be changed, need not be saved.
r1	Always zero in C code, **r1** = 0.	Callee must return it as zero to C code.
r2 .. **r17**	Local data.	Invariant over a call. Callee-save.
r18 .. **r27**	Local data or scratch.	Can be changed. Caller-save if needed.
r28 .. **r29** = **Y**	Local data. May be frame pointer.	Invariant over a call. Callee-save.
r30 .. **r31** = **Z**	Local data or scratch.	Can be changed. Caller-save if needed.

Stacked local variable access

Since the AVR has no useful **SP**-relative addressing mode, *gcc* often uses the **Y** register as a frame pointer and accesses stacked data using **Y**-relative offset addressing in the **ldd** and **std** instructions. However, the offset range in these instructions is limited to 0 .. 63, so in subprograms with much local data the address of a stacked variable must generally be computed using 16-bit additions.

GCC prologue and epilogue routines

Under the default options GCC generates prologue and epilogue code in-line in the application subprogram. This needs no special action from Bound-T and should not cause any problems in the analysis.

When GCC is given the compiler option *-mcall-prologues* it generates shared prologue and epilogue routines that are invoked from the application subprograms that need them. However, these invocations do not use the normal **call** instruction. To invoke a prologue GCC generates code that stores the return address in the **Z** register and jumps to the prologue. The prologue returns with an indirect jump instruction **ijmp** that jumps to the address in **Z**. Bound-T analyses these jump instructions and the instructions in the prologue as if they were part of the invoking subprogram. Bound-T may issue a warning about the dynamic indirect jump but should be able to resolve this jump.

To invoke an epilogue GCC simply generates a jump to the epilogue routine (as in an optimised tail call). The epilogue routine ends with a **ret** instruction that returns from the application subprogram. Again Bound-T analyses the jump instruction and the instructions in the epilogue as if they were part of the invoking subprogram.

In summary, GCC prologues and epilogues pose no special problems for the analysis. The Bound-T command-line option *-logues=call* has no effect on the analysis of GCC programs because Bound-T does not recognize that prologues or epilogues are involved in these jumps. Prologues and epilogues are always analysed as if the option were *-logues=integrate*.

Stack-usage analysis

Stack-usage analysis works poorly at present for *gcc*-generated code, because the instruction sequences that *gcc* generates to allocate and deallocate stack frames use 16-bit arithmetic with the **SPH**:**SPL** pair in ways that Bound-T currently cannot analyse well.

GCC option -mtiny-stack

This option works under the assumption that the stack usage is never more than 255 TBC octets and that there is no carry from the low octet to the high octet (**SPH**), for example because the stack is allocated starting at a 256-octet boundary (**SPL** initial value is zero).

Note that *gcc* still assumes that the **SP** is 16 bits in size. For example, when *gcc* sets the **Y** register to be the frame pointer it sets **Y** to **SPH**:**SPL**, using both the high and low octet of the **SP**.

Support for this option in Bound-T is under investigation.

GCC switch-case statements

Bound-T cannot currently analyse *gcc* code for switch-case statements if the compiler has generated a table of address for use with the **ijmp** instruction. Such tables tend to appear when the switch-case statement has a densely numbered set of case values.

Support for such switch-case code is planned.

7 WARNINGS AND ERRORS FOR AVR

7.1 Warning messages

The following lists the Bound-T warning messages that are specific to the AVR or that have a specific interpretation for this processor. The messages are listed in alphabetical order. The Bound-T Reference Manual [1] explains the generic warning messages, all of which may appear also when the AVR is the target. The Bound-T Assertion Language Manual [3] explains the warning messages from the assertion parser.

The AVR-specific warning messages refer mainly to unsupported or approximated features of the AVR.

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 us for an explanation of any Bound-T output that seems obscure.

Table 16: Warning Messages

Warning Message		Meaning and Remedy
Ambiguous chaining with *N* possible chained operations.	*Reasons*	While trying to "chain" two 8-bit operations to a single 16-bit operation, Bound-T finds $N > 1$ possible 16-bit operations that could represent the pair of 8-bit operations, and therefore leaves the pair unchained.
	Action	Please report to Tidorum.
BREAK instruction taken as no-operation	*Reasons*	There is a **break** instruction at this point in the AVR program. The "execution" time of a **break** instruction depends on the occurrence of external events; Bound-T does not include this time in the analysis.
	Action	Understand that the execution-time bound computed for this part of the program does not include the time the processor spends in **break**.
Call defines virtual mood *C* but subprogram has *S*.	*Reasons*	At the present call in the program, there is a conflict between the calleer and the callee regarding the analysis of virtual-function calls. The call specifies the analysis method defined by the "mood" *C*, while the callee specifies *S*.
	Action	Please report the problem to Tidorum.
Callees unknown for virtual call: *class* | *function*	*Reasons*	According to the UBROF file, the indirect (dynamic) call instruction at this point in the program is a call of the C++ virtual *function* through a pointer to an object of the compile-time *class*, but Bound-T is unable to find the possible real functions that might be called. The call is left as an unresolved (open) dynamic call.
	Action	Ask Tidorum to analyse the problem. As a work-around, use assertions to list the possible callees.

Warning Message		Meaning and Remedy
Code address wraps around from *A* to *B*	*Reasons*	The address of the next instruction would normally be *A*, but *A* is outside the range of code addresses in the chosen AVR device. The actual address is therefore *A* modulo code-memory size, which is *B*, a valid address for this device.
	Action	Verify that you have chosen the correct *-device* option. For some small AVR devices compilers deliberately use **rjmp** instructions that wrap around in this way.
Immediate octet *U* used signed = *S*	*Reasons*	When modelling an 8-bit operation between a register and an immediate 8-bit operand, Bound-T chose to model the immediate number as the signed (negative) quantity *S* rather than the large unsigned quantity *U*.
	Action	This information can help to understand the results of loop-bound analysis. Use the option *-warn no_sign* to suppress these warning messages.
Large combined 16-bit literal *U* taken as signed = *S*	*Reasons*	In its analysis of the program, Bound-T has combined ("chained") two 8-bit operations between two registers and two 8-bit immediate operands (numbers) into a 16-bit operation between this register pair and the 16-bit number composed of the two 8-bit numbers. However, if the 16-bit number is interpreted as an unsigned number *U*, it is so large that Bound-T chooses to interpret it as the negative number *S*.
	Action	This information can help to understand the results of loop-bound analysis. Use the option *-warn no_sign* to suppress these warning messages.
Loading data from segment of type *T*	*Reasons*	The UBROF file contains a segment that is marked to contain constant data of type T. Bound-T loads this data into its program model and assumes that the data are not altered by the program as it runs.
	Action	Check that the program indeed does not change the data in this segment.
No STABS symbols found	*Reasons*	The ELF target program (executable file) contains no STABS symbol-table (debug information in STABS form). Bound-T will try to use the ELF symbol-table instead.
	Action	If a STABS table is necessary, try to find the options for your cross-compiler or linker that make them place STABS information in the ELF file.
Offset of parameter *P* exceeds stack height at call, *H*	*Reasons*	While analysing the parameters passed in this call, Bound-T has found a stacked parameter *P* (shown here in the machine-level form) that has such a large offset from the start of the callee's stack frame that it cannot be a stack location within the caller's stack frame, which only contains *H* octets at the point of the call. The form of *P* shows if it lies in the processor stack or in the auxiliary software stack.
	Action	Ask Tidorum to study the problem.

Warning Message		Meaning and Remedy
Program implies auxiliary stack *A* but option forces stack *B*	*Reasons*	The form or content of the executable target program file (*eg*. UBROF) suggests that the program uses an auxiliary software-defined stack of type *A* (*eg*. using the **Y** register and growing downwards), but a command-line option forces Bound-T to assume a stack of the different type *B* in the analysis.
	Action	Check that the command-line option is correct.
Program implies protocol *A* but option forces protocol *B*	*Reasons*	The form or content of the executable target program file (*eg*. UBROF) suggests that the program uses calling protocol A (*eg*. the IAR protocol), but a command-line option forces Bound-T to use protocol *B* in the analysis.
	Action	Check that the command-line option is correct.
Skipping non-Absolute segment of type *T*	*Reasons*	The UBROF program file being loaded contains a code segment of type *T* that is not an "absolute" (relocated) segment. Bound-T skips (ignores) this code.
	Action	Check that the UBROF file contains linked, executable code, not merely compiled or assembled relocatable code. Bound-T cannot analyse relocatable code.
Skipping segment of type *T*	*Reasons*	The UBROF program file being loaded contains a segment of type *T* that is not useful to Bound-T and is therefore skipped (ignored).
	Action	If you think that this segment is relevant to the analysis, please inform Tidorum about the problem.
SLEEP time not included in analysis	*Reasons*	There is a **sleep** instruction at this point in the AVR program. The "execution" time of a **sleep** instruction depends on the occurrence of external events; Bound-T does not include this time in the analysis.
	Action	Understand that the execution-time bound computed for this part of the program does not include the time the processor spends in **sleep**.
SPM instruction taken as no-operation.	*Reasons*	There is an **spm** (Store Program Memory) instruction at this point in the AVR program. Bound-T's model of **spm** instructions is incomplete in two respects: Firstly, Bound-T assumes that the program under analysis is fixed (not variable), but **spm** changes the contents of program memory (flash). Secondly, the execution time of an **spm** instruction is variable, but Bound-T currently assumes only one cycle for this time.
	Action	Firstly, check that the **spm** does not modify the parts of the program that are being analysed. Secondly, understand that the execution-time bound computed for this part of the program may be underestimated, depending on the AVR device and the function of this **spm** instruction.
Starting IAR switch handling.	*Reasons*	The program contains a switch-case statement for which the IAR compiler has generated a call to a library routine and a switch table, and Bound-T is starting the partial evaluation of the switch handler routine. Refer to the *-switch_eval* option in Table 2.

Warning Message		Meaning and Remedy
	Action	None, but the flow-graph for this subprogram may be larger than expected because it will contain the partially evaluated (expanded) residual code of the switch handler.
Value *V* exceeds 16 bits, *R* becomes opaque.	*Reasons*	During the partial evaluation of a switch handler, the apparent value *V* of a 16-bit register-pair *R* exceeds the 16-bit range for some (unknown) reason. The evaluation continues with an unknown value in *R*.
	Action	The results of the partial evaluation may be in doubt. For safety, change the compiler options or the program to avoid this kind of switch-case code.

7.2 Error messages

The following lists the Bound-T error messages that are specific to the AVR or that have a specific interpretation for this processor. The messages are listed in alphabetical order. The Reference Manual [1] explains the generic error messages, all of which may appear also when the AVR is the target. The Bound-T Assertion Language Manual [3] explains the error messages from the assertion parser.

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 us for an explanation of any Bound-T output that seems obscure.

Table 17: Error Messages

Error Message		Meaning and Remedy
Aborting switch-handler evaluation after *N* steps in the host flow graph.	*Problem*	The analysis of an IAR switch handler routine is aborted because the number *N* of steps (instructions) generated in the flow-graph of the subprogram that contains the switch-case statement has become larger than expected.
	Reasons	The structure of the switch handler may be such that the partial evaluation method [9] does not detect the end of the switch table, and therefore continues evaluating data after the switch table. Alternatively, there may be so many cases in the switch-case structure that the default limit on the size of the flow-graph is too small.
	Solution	Change the program or the compiler options to avoid this kind of switch-case code, or try with a larger value of the Bound-T option *-switch_steps*. If neither works, contact Tidorum.
Calling protocol is not defined	*Problem*	Bound-T cannot analyse the program because the calling protocol is not defined.

Error Message		Meaning and Remedy
	Reasons	The target program file does not specify the calling protocol that the program uses, nor was the protocol specified with a command-line option.
	Solution	Specify the protocol with a command-line option. See Table 5.
Cannot determine executable file type	*Problem*	Bound-T could not find out the type (COFF, ELF, UBROF) of the executable target program file named on the command line, and the type was not specified with a *-format* option.
	Reasons	The file is not a COFF, ELF or UBROF file; or is damaged; or uses a variant of COFF, ELF or UBROF that Bound-T does not support.
	Solution	Get an executable file in a form that Bound-T suppports. If you are sure of the file format, try to use the *-format* option.
Cannot read file	*Problem*	Bound-T was unable to open and read the executable target program file.
	Reasons	The file's permissions do not allow reading, or the file-name identifies an object that is not an ordinary file (a directory name, for example).
	Solution	Change the file permissions or correct the file-name.
Code address *A* exceeds memory size *M* octets; wrapped to *B*	*Problem*	The target address A of a call or jump instruction is beyond the size of the code memory, *M* octets, in the chosen AVR device. Bound-T uses the address *A* mod *M*, which is *B*.
	Reasons	The program was probably compiled and linked for an AVR device with a larger code memory.
	Solution	Check and correct the *-device* option, or recompile and relink the program for the correct device.
Extended Indirect Jump/Call is not supported	*Problem*	The program contains an **eijmp** or **eicall** instruction. Bound-T does not yet support these instructions so it will leave the jump or call unresolved.
	Reasons	The program is written in that way.
	Solution	Avoid using these instructions or analyse the program in parts, then add up the execution-time bounds for the parts.
File not found	*Problem*	Bound-T could not open the executable file named on the command line because there is no such file.
	Reasons	The file-name was mistyped; perhaps a directory name is missing; or perhaps some directory included in the file-name does not permit access.
	Solution	Correct the file-name or change directory permissions.
Ignoring asserted “virtual” values (must be single value 0 .. 1).	*Problem*	An assertion defines the value of the property “virtual” (see Table 10) but allows multiple values or values outside the valid range.
	Reasons	The assertion is in error.

Error Message		Meaning and Remedy
	Solution	Correct or remove the assertion.
Illegal Load Indirect with pointer update	*Problem*	The program contains an **ld** instruction that uses auto-increment or auto-decrement with a destination register that is part of the pointer being auto-modified. The result of such an instruction is undefined, according to [4].
	Reasons	The target program is written in this way. See also the other reasons listed for the error message "Instruction not recognized".
	Solution	Correct the target program. See also the other solutions listed for the error message "Instruction not recognized".
Illegal Load Program with pointer update	*Problem*	The program contains an **lpm** instruction with auto-increment and a destination register that is part of the **Z** pointer. The result of such an instruction is undefined, according to [4].
	Reasons	The target program is written in this way. See also the other reasons listed for the error message "Instruction not recognized".
	Solution	Correct the target program. See also the other solutions listed for the error message "Instruction not recognized".
Incorrect software stack definition: -swstack=*S*	*Problem*	The part *S* in the *-swstack* option is not of the expected form.
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 2.
Incorrect switch_steps value: -switch_steps=*S*	*Problem*	The part *S* in the *-switch_steps* option is not of the expected form.
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 2.
Instruction not recognized	*Problem*	The program contains an instruction word that Bound-T cannot decode as a valid AVR instruction.
	Reasons	The target program file may be damaged; it may use an extended AVR instruction set that Bound-T does not support; it may contain code for some other microprocessor family; or Bound-T's program-flow analysis may be in error, making Bound-T try to decode some program memory content that is not meant to be decoded as AVR instructions (for example, string constants stored in flash).
	Solution	Use the option *-trace decode* to see where the error occurs. Contact Tidorum if the error seems to be in the program-flow analysis.
No -device was specified	*Problem*	Bound-T cannot analyse this program because the AVR device (chip or model) is not known.
	Reasons	There device was not specified on the command line.
	Solution	Use the *-device* option. See Table 7.

Error Message		Meaning and Remedy
No instruction loaded at this address	*Problem*	According to Bound-T's analysis, the program fetches an instruction from a program memory address that is blank; that is, the target program file does not load any code at this address.
	Reasons	The target program file is incomplete; or the program itself stores something at this address at run-time with the **spm** instruction; or the command line specifies a root-subprogram address that points to a blank part of the program memory; or Bound-T's program-flow analysis is in error.
	Solution	Prepare a complete target-program file; avoid self-modifying code; give the correct root-subprogram address; or contact Tidorum if the error seems to be in the program-flow analysis.
Odd octet address *A* cannot be an instruction address	*Problem*	The command-line or an assertion specifies *A* as the octet address of a subprogram or an instruction, but this is impossible because *A* is an odd number while all instructions lie at even octet addresses.
	Reasons	The command-line or assertion is in error.
	Solution	Correct the command-line or the assertion.
Odd register number *R* for word variable.	*Problem*	The UBROF symbol-table entry for a word-sized (16-bit) variable locates the variable in a register pair that starts with an odd register number *R*.
	Reasons	The IAR compiler uses registers in a way that conflicts with Bound-T's assumptions.
	Solution	Use a version of the IAR compiler that Bound-T supports, or report the problem to Tidorum.
Patching is not implemented for this target.	*Problem*	The command-line contains a *-patch* option (a general Bound-T option [1]) with a non-empty patch file.
	Reasons	Patching is not implemented in Bound-T for the AVR.
	Solution	Remove the *-patch* option from the command-line, or ask Tidorum to implement this option for the AVR.
Return by offset *X* from strange state *S*	*Problem*	In this call, the callee has been asserted to return not to the normal return point, but to an address offset by *X* octets from the normal return point. However, the modelled state *S* of the processor at the return point is not a normal state, making the use of an offset return suspect.
	Reasons	None known.
	Solution	Please report the problem to Tidorum.
Unexpected end of COFF file *or* Unexpected end of ELF file *or* Unexpected end of UBROF file	*Problem*	While Bound-T was reading the COFF (or ELF or UBROF) target program the file ended at an unexpected place.
	Reasons	The executable file is damaged or uses a variant of COFF (or ELF or UBROF) that Bound-T does not support.

Error Message		Meaning and Remedy
	Solution	Get an executable file in a form that Bound-T suppports.
Unexpected end of file	*Problem*	While Bound-T was reading the target program the file ended at an unexpected place.
	Reasons	The target program file is damaged or uses a variant of COFF, ELF or UBROF that Bound-T does not support.
	Solution	Get a a target program executable file in a form that Bound-T suppports.
Unknown AVR calling protocol: -protocol=*P*	*Problem*	The part *P* in the *-protocol* option is not the name of a supported calling protocol.
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 5.
Unknown -endian vaue: -endian=*E*	*Problem*	The part *E* in the *-endian* option is not "little" or "big".
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 2.
Unknown -logues value: -logues=*L*	*Problem*	The part *L* in the *-logues* option is not one of the supported choices.
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 2.
Unknown -format value: -format=*F*	*Problem*	The part *F* in the *-format* option is not the name of an executable format that Bound-T for AVR supports.
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 3.
Unresolved indirect exit-jump from IAR switch handler.	*Problem*	While analysing an IAR switch handler using the partial-evaluation method [9] Bound-T has found an indirect jump that should exit (terminate) the handler routine, but the partial evaluation is unable to compute the target address.
	Reasons	The switch handler routine is too complex for this method of analysis.
	Solution	Change the program or the compilation options to remove this kind of switch-case code. If that is not possible, contact Tidorum.
Unknown switch-offset value: -switch_offset=*V*	*Problem*	The part *V* in the *-switch_offset* option is not one of the supported choices.
	Reasons	Mistyped command-line option.
	Solution	Correct the command-line option. See Table 2.

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