Bound-T time and stack analyzer

Application Note

MCS®-51 (8051) Family

TR-AN-8051-001
Issue 3
2010-02-09
Tidorum Ltd.

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

The first issue of this document was written at Space Systems Finland Ltd by Ville Sipinen.

The document is currently maintained by Niklas Holsti at Tidorum Ltd.

Document reference:	TR-AN-8051-001
Document issue:	Issue 3
Document issue date:	2010-02-09
Bound-T version:	4a5
Web location:	*http://www.bound-t.com/app_notes/an-8051.pdf*

Credits:
This document was created with the 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 issue of this document please refer to the web address *http://www.bound-t.com/app_notes/an-8051.pdf*.

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 (+3 hours in the summer) 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* (please include the word "Bound-T" in the Subject line)

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 (*http://www.graphviz.org/*).

Contents

Index of Tables

Document change log

Issue	Section	Changes
3	-	Change log started.
3	1.3	Updated URLs to manuals. Added ref. 17 (OMF2 format), 24 (CC2510 device), 25 (nRF24E1 device), and 26 (nRF24LE1 device).
3	1.5	Added some acronyms.
3	2.1	Added OMF2 as accepted input format.
3	2.2	Added new devices CC2510, nRF24E1, nRF24LE1, and OMF2 support. Added device-specific option tables. Added tables of 8051-specific items for the *-trace* and *-warn* options.
3	3.1	Added address offsets, instruction roles, and property names to the list of target-specific assertion-syntax elements.
3	3.3	Added section on code-address offsets.
3	3.4	Updated description of how DPTR is computed from DPL and DPH.
3	3.6	Added section on instruction roles.
3	3.7	Noted that the "returns" property can be overridden by assertions on instruction roles.
3	5	Updated and focused on features not fully modelled.
3	6.1	Updated to consider instruction role assertions.
3	7.1	Added a note on meaning of "supports a compiler".
3	7.3	Updated to show OMF2 support.
3	7.5	Added section on "other compilers".
3	8.1, 8.2	Updated for evolution in warning and error messages.

1 INTRODUCTION

1.1 Purpose and scope

Bound-T is a tool for computing bounds on the worst-case execution time and stack usage of real-time programs; see references [1, 2]. There are different versions of Bound-T for different target processors. This Application Note supplements the Bound-T User Guide and Reference Manual with information specific to the target processor architecture usually known as the "8051" although it was introduced with the name MCS-51 [6]. We explain how Bound-T models the architecture of the 8051 processor and how to use Bound-T to analyse programs for this processor.

Some information in Chapters 6 and 7 of this Application Note applies only when the target-program executable is generated with specific compilers. These chapters discuss the properties of the IAR Systems C/C++ compiler [9, 10], the Keil C compiler [11, 12], and the SDCC C compiler [13, 14] and how Bound-T can or cannot analyse code that those compilers generate. Other compilers may be addressed in separate Application Notes.

The 8051 architecture is remarkable for its longevity and for the large and still growing number of implementations (chips, devices), variants and extensions from several sources. Some variants extend the "core" instruction set or the register set. Some variants accelerate execution by using fewer clock cycles per instruction. There is also a large spectrum of devices with different amounts of on-chip memory and different sets of on-chip peripherals. However, those device features are not relevant to execution time analysis.

Bound-T supports a number of 8051 chips but not all of them. Some chip parameters can also be defined by the user, through options and assertions.

1.2 Overview

How it's done

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 and constructs the control-flow graph with its branches, calls and loops. Bound-T (partially) interprets the loop arithmetic to find the "loop-counter" variables that control the loops, such as n in "*for* ($n = 1$; $n < 20$; $n{+}{+}$) { ... }".

Bound-T analyses the initial values and steps of the loop-counter variables together with the loop termination condition to compute 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.

Necessary earlier knowledge

To make full use of the information in this Application Note you should already be familiar with the register set and instruction set of this processor, as presented in reference [6]. You should also be familiar with the general principles and usage of Bound-T, as described in the Bound-T User Guide [1]. The user manual also contains a glossary of terms, many of which will be used in this Application Note. The Bound-T Reference Manual [2] is also useful background information.

Contents

After this introductory chapter the remainder of this Application Note consists of a user guide part and a reference part. The user guide part consists of chapters 2 and 3:

- Chapter 2 shows how to use Bound-T for the 8051. It lists and explains the command arguments and options that are wholly specific to the 8051 or that have a specific interpretation for this processor. It also explains the Bound-T outputs specific to the 8051.
- Chapter 3 addresses the user-defined assertions on target program behaviour and explains the possibilities and limitations in the context of the 8051 and the development tools we consider here: IAR, Keil and SDCC.

The rest of the Application Note forms the reference part:

- Chapter 4 describes the main features of the 8051 architecture and how Bound-T models them in general.
- Chapter 5 defines in detail the set of 8051 instructions and registers that is supported by Bound-T.
- Chapter 6 explains how Bound-T models and analyses subprogram calls and parameters, focussing on the procedure calling standard (calling protocols, parameter-passing methods).
- Chapter 7 discusses the cross-compilers that Bound-T supports for the 8051: the IAR, Keil, and SDCC compilers. The chapter explains which language and compiler features are currently supported and which are not.
- Chapter 8 lists all 8051-specific warnings and error messages that Bound-T can issue and explains the possible reasons for each.

1.3 References

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

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

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

[4] find-marks User Manual.
Tidorum Ltd, Doc. ref. TR-UM-004.
http://www.bound-t.com/manuals/find-marks-manual.pdf.

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

[6] 8-bit Embedded Controller Handbook
Intel © 1990.

[7] The C51 Primer.
Phaedrus Systems, Chris Hills (ed.), Edition 3.6, 17 January 2006.
http://www.phaedsys.demon.co.uk/chris/papers/QuEST4_1.pdf.

[8] C8051F120/1/2/3/4/5/6/7, C8051F130/1/2/3: Mixed Signal ISP Flash MCU Family.
Silicon Laboratories, Rev 1.3 8/04.

[9] IAR Systems. *http://www.iar.com/*.

[10] 8051 IAR C/C++ Compiler Reference Guide for the MCS-51 Microcontroller Family. IAR Systems, part number C8051-3, third edition (July 2005).

[11] Keil – an ARM company. *http://www.keil.com/*.

[12] Keil Cx51 User's Guide.
CHM file in C51 version 8.09.

[13] Small Device C Compiler (SDCC). *http://sdcc.sourceforge.net/*.

[14] SDCC Compiler User Guide.
SDCC 2.7.0, 2007-05-29, Revision 4818.

[15] External Product Specification for the MCS-51 Object Module Format.
Intel Corporation, V5.0, Sept 05, 1982.

[16] Additions to the 8051 Object Module Format (OMF-51).
Keil Elektronik GmbH, 05/07/2000.

[17] External Product Specification for the Object Module Format: 251/MX51 Specification (OMF2 Format).
Keil Software, Rev 2.21, 30-Jun-2006.

[18] AOMF with Keil C51 extensions as input to Bound-T.
Tidorum Ltd, Doc. ref. TR-TN-AOMF-001.
http://www.bound-t.com/tech_notes/tn-aomf.pdf.

[19] Intel® Hex as input to Bound-T.
Tidorum Ltd, Doc. ref. TR-TN-IHEX-001.
http://www.bound-t.com/tech_notes/tn-ihex.pdf.

[20] CDB File Format.
Lenny Story, SDCC Development Team, 2003-03-21.

[21] ASxxxx Assemblers and ASLINK Relocating Linker.
Alan R. Baldwin, Kent State University, Version 2.0, August 1998.

[22] CDB from SDCC as input to Bound-T.
Tidorum Ltd, Doc. ref. TR-TN-CDB-001.
http://www.bound-t.com/tech_notes/tn-cdb.pdf.

[23] Analysing Switch-Case Tables by Partial Evaluation.
Niklas Holsti, 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007), Pisa, Italy, July 3, 2007.
http://www.tidorum.fi/bound-t/reports/wcet2007/simcase.pdf
or *http://www.irit.fr/wcet2007/wcet07_proceedings.pdf*.

[24] CC2510F8: 2.4 GHz Radio Transceiver, 8051 MCU and 8 kB Flash memory.
http://focus.ti.com/docs/prod/folders/print/cc2510f8.html

[25] nRF24E1 Transceiver / MCU / ADC — Nordic Semiconductor nRF24E1 — System on Chip with 8051 MCU.
http://www.nordicsemi.com/index.cfm?obj=product&act=display&pro=79

[26] nRF24LE1 – Ultra low power wireless System-on-Chip solution
http://www.nordicsemi.com/index.cfm?obj=product&act=display&pro=95

1.4 Typographic conventions

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

register	The name of an 8051 register embedded in prose.
instruction	An 8051 instruction.
-option	A command-line option for Bound-T or other tools. In some tables the style -option is used for clarity.
symbol	A mathematical symbol or variable.
`text`	Text quoted from a text / source file or command.
identifier	An identifier from a program.

1.5 Abbreviations and acronyms

See also reference [1] for abbreviations specific to Bound-T and reference [6] for the mnemonic operation codes and register names of the 8051.

A	The main accumulator register.
AC	The auxiliary carry flag (in the **PSW**).
ACC	A synonym for the **A** register.
AOMF	Absolute Object Module Format. See references [15], [16], and [18].
AOMF2	Absolute subset of OMF2. See OMF2.
B	The secondary accumulator register; the **B** register.
BCD	Binary Coded Decimal.
C	Carry Flag (in the **PSW**)
CDB	C DeBug (?) format [20] used with SDCC [13].
DPTR	The Data Pointer Register, a 16-bit register in the Intel-8051, used to address data in the external memory or in the code memory.
EC++	Embedded C++.
IDE	Interactive Development Environment.
IHex	Intel Hex (format) [15].
Ko	Kilo-octet = 1024 octets = 2^{10} octets.
LSB	Least Significant Byte.
Mo	Mega-octet = 2^{20} octets.
MSB	Most Significant Byte.
OMF2	Object Module Format 2. See reference [17].
OV	Overflow flag (in the **PSW**).
PSW	Program Status Word.
RS0	The first register-bank selector bit in the **PSW**.
RS1	The second register-bank selector bit in the **PSW**.
SDCC	Small Device C Compiler [13].
SP	The Stack Pointer register.

SFR	Special Function Register.
TBA	To Be Added.
TBC	To Be Confirmed.
TBD	To Be Determined.
UBROF	Universal Binary Relocatable Format, defined by IAR Systems [9].
WCET	Worst-Case Execution Time.

2 USING BOUND-T FOR 8051

2.1 Input formats

Bound-T for the 8051 can read three executable program file formats:

- The Absolute Object Module Format (AOMF) as originally defined by Intel [15], with extensions for the Keil compiler [16]. See [18] for details on Bound-T support for AOMF.
- The absolute subset of the OMF2 format as defined by Keil [17] for the C51/CX51 compiler and the LX51 linker.
- The Universal Binary Relocatable Object Format (UBROF) as defined by IAR Systems [9] and used by the IAR compiler [10]. This is a proprietary closed format.
- The Intel Hex format, a textual representation of memory images emitted by the SDCC compiler [21, section 2.7]. See [19] for details on Bound-T support for for Intel Hex. The symbolic debugging information can be read from a separate file in the CDB format [20].

Bound-T can usually detect the format automatically from the contents of the file itself. The format can also be set by a command-line option (*-form*).

Bound-T for the 8051 can also read three types of additional input files for Bound-T:

- assertion files (option *-assert*) as defined in [3],
- mark definition files (option *-mark*) as defined in [4], and
- symbol files (option *-symbols*) as defined in [2].

Patch files (option *-patch* [2]) are not yet supported for the 8051 target.

2.2 Command arguments and options

Command line form

The generic Bound-T command format, options, and arguments as explained in the Reference Manual [2] apply without modification to the 8051 version of Bound-T. The command line usually has the form

```
boundt_8051 options program-file root-subprogram-names
```

For example, to analyse the execution time on the ordinary 8051 processor of the main sub-program in the AOMF executable program file prog.omf under the option *-rxx*, the command line is

```
boundt_8051 -device=8051 -rxx prog.omf main
```

Naming root subprpograms

Root subprograms can be named by the link identifier, if present in the program symbol-table, or by the entry address in hexadecimal form with a trailing letter “H” and optionally prefixed with “C:”. Thus, if the entry address of the main subprogram is 4A0 (hex), the above command can also be given as

```
boundt_8051 -device=8051 -rxx prog.omf 4A0H
```

Some hexadecimal addresses may match link identifiers. For example, the program may contain a subprogram with the link identifier A4H. To force Bound-T to use the root subprogram starting at address A4H, instead of the subprogram named A4H, prefix the address with a zero or the string "C:", writing it 0A4H or C:A4H or indeed C:0A4H (assuming that no subprogram has a link identifier of that form). For example:

```
boundt_8051  -device=8051  -rxx  prog.omf  C:A4H
```

Options in general

All the generic Bound-T options apply (but the *-patch* option has no effect). There are additional 8051-specific options as explained below. The generic option *-help* makes Bound-T list all its options, including the target-specific options.

Note that a target-specific option must be written as one string with no embedded blanks, so the option-name and its parameter, if any, are contiguous and separated only by the equal sign (=) but not by white space. For example, the form "*-reg_bank=1* " is correct but "*-reg_bank = 1* " is not.

For HRT analysis please refer to the separate Bound-T manual discussing the HRT mode [5]. There are no specific considerations or options for HRT analysis on the 8051.

The explanation of the 8051-specific options is grouped below as follows:

- Target device selection options and device-specific options.
- Compiler selection options and compiler-specific options.
- Input format selection options.
- Instruction modelling options.
- 8051-specific items for the generic *-trace* option.
- 8051-specific items for the generic *-warn* option.

Target device selection options

There are many variants of the 8051 processor. You must tell Bound-T which kind of "8051" processor the target program is meant for, unless the default of a "standard" 8051 is valid.

Use the option *-device=name* to select the target processor by its *name*. This option can also be written *-device name* or even just *-name* if there is no confusion with other options.

Table 1 below lists the 8051 devices (models, chips, derivatives) that can be selected with the *-device* option. If this option is not used Bound-T assumes a "standard" 8051. Note that each of these "device names" typically corresponds to a whole family of 8051 devices that are equivalent for Bound-T because they have the same instruction set and instruction timing and differ only in the amount of memory or the set of on-chip peripherals.

Please do not hesitate to ask Tidorum about support for the particular 8051 device that you would like to use.

Table 1: Supported 8051 devices and families

Option	Device	Properties and their support in Bound-T
-device=8051	"Standard" MCS51/8051 core	See ref. [6]. Execution time unit: machine cycle (12 clock cycles).

Option	Device	Properties and their support in Bound-T
-device=cc2510	Texas Instruments CC2510	See ref. [24]. Instruction cache (flash cache) is assumed to always hit. Access to other memory spaces (internal data, SFR, code) through the non-standard "unified" XDATA mapping is not modelled. Execution time unit: clock cycle.
-device=cip-51	Cygnal CIP-51 series core	See ref. [8]. Dynamic changes in program memory bank are not supported. The analysed part of the program is assumed to be contained in the lower 32 Ko bank and in one of the upper 32 Ko banks. SFR paging is not supported. A branch-cache miss is assumed for all branches. Execution time unit: clock cycle.
-device=nrf24e1	Nordic Semiconductor nRF24E1	See ref. [25]. Dual DPTRs not modelled. External data access time (**movx** instruction) is assumed to be 3 cycles but can be changed by command-line option. Dynamic changes through the **CKCON** SFR are not modelled. Execution time unit: machine cycle (4 clock cycles).
-device=nrf24le1	Nordic Semiconductor nRF24LE1	See ref. [26]. Dual DPTRs not modelled. Write to flash not modelled. Execution time unit: clock cycle.

After selecting a device with the *-device* option, you can set some device-specific options as described in the tables below for the CIP-51 and nRF24E1 devices. There are no device-specific options for the other devices. Note that the device-specific options, if any, must come *after* the *-device* option on the command line.

Table 2: Device-specific options for CIP-51

Option	Meaning and default value	
-no_branch_cache	*Function*	The device has no branch cache, or branch cache misses should be ignored in the execution-time analysis.
	Default	Each taken branch and return is assumed to cause a branch-cache miss, which causes a 4-cycle delay.

Table 3: Device-specific options for nRF24E1

Option	Meaning and default value	
-movx=*C*	*Function*	Each **movx** instruction is assumed to take *C* machine cycles. The value *C* should equal or exceed two plus the value that the program configures in the **CKCON** register, bits 2-0.
	Default	The default is *-movx=3*, matching the reset value of **CKCON** bits 2-0.

Compiler selection options

There are several cross-compilers that can generate 8051 programs from source code in C or assembly language. For C compilers Bound-T may need to know which compiler has generated the program under analysis because different compilers use different conventions for parameter passing and register saving and restoring. Moreover, some compilers have peculiar library functions or code-generation styles that require special analysis.

Use the option *-compiler=name* to tell Bound-T which cross-compiler generated the target program. This option can also be written just as *-name* if there is no confusion with other options. Table 4 below lists the supported cross-compilers that can be selected with the *-compiler* option. If this option is not used Bound-T guesses the compiler based on the format of the target program as shown in the last column of the table.

Table 4: Supported cross-compilers

Option	Compiler	Assumed if program format is
-compiler=iar	IAR Systems [9]	UBROF
-compiler=keil	Keil [11]	AOMF or AOMF2
-compiler=sdcc	SDCC [13]	Intel Hex

After selecting the compiler the compiler-specific options become available as explained below. Note that compiler-specific options cannot be used if Bound-T itself (by default) selects the compiler based on the program-file format. Compiler-specific options, if any, must come after the *-compiler* option on the command line.

Options for the IAR compiler

At present there are no options specific to the IAR compiler.

Options for the Keil compiler

At present there are no options specific to the Keil compiler.

Options for the SDCC compiler

Table 5 below explains the command-line options that are specific to the SDCC compiler. These options can only be set after the SDCC compiler is chosen with the option *-compiler=sdcc* or *-sdcc*.

Table 5: Options for the SDCC compiler

Option	Meaning and default value	
-aslines	*Function*	Includes source-line/code-address connections generated from assembly-language files, in addition to connections generated from C files.

Option	Meaning and default value	
		This option may be useful when parts of the target program are written in assembly language. It lets Bound-T show the source-line numbers in the assembly-language source files for subprograms, loops and other program parts. However, the assembly-level line-number information may be confusing and unnecessary for programs written in C.
	Default	The default is *-no_aslines*.
-no_aslines	*Function*	Omits source-line/code-address connections generated from assembly-language files. Connections generated from C files are still included.
	Default	This is the default.
-cdb=*filename*	*Function*	Names the CDB file that contains the SDCC debugging information for this target program.
	Default	By default Bound-T looks for a CDB file that has the same name as the target program file but the suffix ".cdb".
-defxu8	*Function*	When a CDB symbol is defined by a Linker record but has no compiler record that would give its type and size, assume that the symbol represents an unsigned 8-bit octet in external data memory. The problem of missing compiler-generated CDB records occurs, for example, for C variables that are declared with the "__pdata" modifier to place them in the paged part of the external memory. See also the option *-warn cdb_def*.
	Default	This assumption is the default. However, if this assumption is false for some such CDB symbol then you should not use this symbol in assertions because the assertion may not work correctly.
-no_defxu8	*Function*	Ignore any CDB symbol that is defined by a Linker record but has no compiler record that would give its type and size.
	Default	This is not the default. See *-defxu8*.
-warn [no_]cdb_def	*Function*	Enables or disables the warning message that reports that a CDB symbol has been given an assumed type, size and location because the symbol has no compiler record.
	Default	This warning is disabled by default.

Input format selection options

Bound-T can read 8051 target programs in several file formats. By default Bound-T tries to deduce the format from the contents of the file itself, but if this fails you can use option *-form=name* to tell Bound-T to assume the named file format.

Table 6 below lists the target-program file formats that can be selected with the *-form* option. The table also shows which cross-compiler Bound-T assumes for this format unless the *-compiler* option overrides this assumption.

Table 6: Supported target program file formats

Option	Format	Assumed compiler	See also
-form=aomf	Absolute Object Module Format [15] with Keil extensions [16]	Keil	[18]
-form=aomf2 -form=omf2	Absolute OMF2 format [17]	Keil	
-form=ihex	Intel Hex [21, section 2.7] perhaps with a CDB file [20]	SDCC	[19, 22]
-form=ubrof	IAR Universal Binary Relocatable Format	IAR	

Instruction modelling options

Bound-T analyses the target program by *reading* the instructions from the executable file, *decoding* the binary form of each instruction into its logical representation in the 8051 instruction set, and then *modelling* the computation that the instruction performs. The modelling part is controlled and guided by command-line options for several reasons, including the following:

- The model of a given 8051 instruction often depends on global settings, such as the choice of register bank, that could only be computed by a detailed analysis of the whole program, if at all. Such whole-program analysis is impractical for Bound-T. Therefore command-line options are provided to specify such global settings.
- Some instructions are complex to model in general, but may be much simpler to model if it is known that the target program uses them in a restricted or specific way. One example is the **ret** instruction which sets the Program Counter (PC) to a 16-bit value popped off the stack. Normally the **ret** instruction is used to return from the current subprogram (the popped value is the return address) and as such is simple to model. However, sometimes **ret** is used as a dynamic jump, by pushing the computed destination address onto the stack and executing **ret** to set the PC to the computed address. This is much harder to model. Thus, there is a command-line option that tells Bound-T which model to use for **ret**.

Table 7 below lists and explains the options that control instruction modelling.

Table 7: Instruction modelling options

Option	*Meaning and default value*	
-[no_]bit_mix	*Function*	Enables or disables the interaction of operations on bits (bit-addressed boolean data) with operations on the "host" octets that contain those bits. Disabling this interaction is strongly discouraged as it may lead to wrong analysis results (an instruction that changes a bit is not modelled as changing the host octet, and vice versa).
	Default	The default is *-bit_mix*.
-[no_]bit_range	*Function*	Enables or disables the explicit constraining of all 1-bit cells to the range 0 .. 1 in the arithmetic model of bit (boolean) instructions. When the explicit constraints are disabled the value of 1-bit cells can only be deduced from the expressions assigned to the cells.
	Default	The default is *-bit_range*.

Option	*Meaning and default value*	
-page=*address*	*Function*	Sets the base *address* (in hex) of "paged data" in external data memory. The high 8 bits of the *address* define the high 8 bits of the external data-memory address for **movx** instructions that use an 8-bit register (**r0** or **r1**) to define the low 8 bits of the address. The given base *address* must be a multiple of 256, that is, the last two hex digits must be "00". Write the *address* using just the hex digits (no "H" suffix and no "X:" prefix).
	Default	*-page=0*
-reg_bank=*b*	*Function*	Sets the number *b* of the register bank to be assumed, where *b* is a number from 0 to 3. Bound-T assumes that the **RS0**-**RS1** bits in the **PSW** are set to this bank.
	Default	Bank zero, as in *-reg_bank=0*.
-returns=dynamic	*Function*	Tells Bound-T to model the return-from-subroutine instruction **ret** as a dynamic transfer of control to the address popped from the stack. After analysis the result can be a dynamic jump to this address or it can be a normal return form the subprogram that contains the **ret** instruction. You can override this command-line option for a subprogram by asserting the "returns" property for that subprogram. See section 3.7.
	Default	The default is *-returns=static*, see below.
-returns=static	*Function*	Tells Bound-T to model the return-from-subroutine instruction **ret** always as a normal return form the subprogram that contains the **ret** instruction. In other words, we assume that the address on the top of the stack when **ret** is executed is always the return address for the current subprogram. You can override this command-line option for a subprogram by asserting the "returns" property for that subprogram. See section 3.7.
	Default	This is the default.
-rxx	*Function*	When Bound-T displays a disassembled instruction, a reference to the register-bank area of the internal data memory is shown as "r<bank><register>" instead of a simple direct address. For example, the address 28 = hex 1C is disassembled as "r34" = bank 3, register 4.
	Default	The default is *-no_rxx*, which see.
-no_rxx	*Function*	When Bound-T displays a disassembled instruction, a reference to the register-bank area of the internal data memory is shown as a simple direct address, not as "r<bank><register>".
	Default	This is the default.
-[no_]sfr_counters	*Function*	Enables or disables the analysis of Special Function Registers (SFRs) as possible loop counters. If enabled Bound-T assumes that the target program may use any SFR as a loop counter (a loop induction variable important for loop termination). If disabled Bound-T assumes that only the SFRs **A** (**ACC**), **B**, **DPL**, **DPH**, and the **DPTR** may be used as loop counters.
	Default	The default is *-no_sfr_counters*.

Option	*Meaning and default value*	
-spch=*amount*	*Function*	Makes Bound-T assume that the execution of a callee subprogram changes the **SP** register by the given *amount*, for all callee subprograms and all calls. If the callee pops as much as it pushes and returns normally it decreases **SP** by 2 octets, corresponding to *-spch=-2* (minus two). If the **SP** stack is used only for return addresses and for saving and restoring registers (as is common) this option can facilitate the analysis.
	Default	The default is to analyse the callee to find the amount of change in **SP**, which can be different for each callee. In contrast, *-spch* asserts that all calls and callees change SP by the given amount.

8051-specific -trace options

Table 8 below lists the 8051-specific "items" that can be used in the generic *-trace item* option to ask Bound-T for more informational outputs.

Table 8: 8051-specific -trace items

-trace item	Traced information
aomf_sym	Symbols defined in AOMF or AOMF2 files.
load	Program-file records (AOMF, AOMF2, UBROF, Intel-Hex) as they are read.

8051-specific -warn options

Table 9 below lists the 8051-specific "items" that can be used in the generic *-warn [no_]item* option to ask Bound-T for more or less warning outputs.

Table 9: 8051-specific -warn items

-warn item	Warnings emitted for
bit_ref	An instruction that stores to an internal data location, using an indirect (dynamically computed) address, which Bound-T resolves to the address of an octet in the bit-addressable memory area. This is worth a warning because the model of such instructions does not include the effect on the individual bits within the host octet.

2.3 Outputs

Basic output format

Most Bound-T outputs, including warning and error messages, follow a common, basic format that contains the source-file name and source-line number that are related to the message. This format is explained in the Bound-T Reference Manual [2].

However, some compilers may not provide all the debugging information, depending on the optimization and debugging options. With such target programs, the Bound-T output will also be reduced, for example source-line numbers may be missing.

Units of measurement

The execution speed in terms of clock cycles per instruction varies a lot for different 8051 devices. Therefore we use one of two different time units depending on the chosen 8051 device. These units are explained in the following table. A specific output line with the keyword "Time_Unit" shows which unit is used.

Table 10: Execution time units

Unit	Definition	Typical devices
Machine cycle	For the 8051 devices that use this unit a machine cycle is the minimum execution time of an instruction and is equal to 12 clock cycles. The execution time of any instruction is a multiple of the machine cycle, thus a multiple of 12 clock cycles.	"Standard" 8051 devices.
Clock tick	One clock cycle using the definition of "processor clock" for the chosen 8051 device. The execution time of any instruction is a multiple of the clock cycle.	"Accelerated" 8051 devices.

Stack usage bounds are always given in octets for all 8051 devices.

Outputs specific to the 8051

Bound-T for the 8051 emits additional output lines as explained in Table 11 below.

As explained in the Reference Manual [2], in each output line a keyword in field 1 identifies the kind of output, fields 2 through 5 identify the program element, and the later fields contain the output information. The table below is ordered alphabetically by the keyword column.

Table 11: Outputs for 8051

Keyword (field 1)	Explanation of fields 6 -
Compiler	*compiler-name* Reports the name of the cross-compiler that was specified with the *-compiler* option or was assumed from the format of the program file.
Device	*device-name* Reports the name of the 8051 device that was specified with the *-device* option. The device determines the instruction set (extensions) and the unit of execution time.
Time_Unit	*unit* Reports the unit of execution time (WCET). See Table 10.

2.4 Example

Here we show a small but complete example of an 8051 program and how Bound-T can find bounds on its execution time and stack usage.

The program

The example program is written in C. The program assumes that the processor has an analog-to-digital converter (digital voltmeter, ADC) that works as follows:

- The SFR bit at address E8H is 0 while a conversion is going on (ADC "busy"), and changes to 1 when it is completed and the digital value is available for reading.
- When bit E8H is 1 the digital value can be read from the SFR at address E3H. When this SFR is read the ADC starts a new conversion automatically and bit E8H becomes 0 until the new conversion is ready.

The example program contains a function ave_adc that computes the average of a number of ADC readings. The main function calls ave_adc twice, first computing the average of 5 readings and then the average of 10 (more) readings; the two averages are then compared. Here is the source code, provided with line numbers in the left margin:

```
1  typedef unsigned char uchar;
2  volatile __sfr __at (0xE3) adc;
3  volatile __bit __at (0xE8) conv;
4
5  uchar ave_adc (uchar count)
6  {
7     unsigned int sum = 0;
8     uchar k;
9     for (k = 0; k < count; k++)
10     {
11        while (conv == 0);
12        sum += adc;
13     }
14     return (uchar) (sum / count);
15 }
16
17
18 int main (void)
19 {
20    return ave_adc (5) == ave_adc (10);
21 }
```

Analysis for execution time

The program contains one unbounded loop: the while-loop that polls the "conversion ready" bit conv within the for-loop in ave_adc. This polling loop must be bounded with an assertion using the maximum number of repetitions computed manually from knowledge of the conversion time. Assume that this computation has been done and that the result is that at most 21 polls are needed. The corresponding assertion is:

```
subprogram "ave_adc"
   loop in loop repeats 21 times; end loop;
end "ave_adc";
```

Assume that the program is compiled and linked into the executable file prog.exe and that the above assertion is written in the file poll.txt. The Bound-T command to find an upper bound on the execution time of the main function is then:

```
boundt_8051 -assert poll.txt prog.exe main
```

The output is the following, when the program is compiled with the SDCC compiler (version 2.7.0) for the "small" memory model:

```
Bound-T 3d2 for 8051
Device:prog.exe::::8051
Time_Unit:prog.exe::::Machine cycles.
Compiler:prog.exe::::SDCC
Loop_Bound:prog.exe:main.c:ave_adc:21-:15
Loop_Bound:prog.exe:main.c:main@20->ave_adc:9-12:5
Loop_Bound:prog.exe:main.c:main@20->ave_adc:21-:15
Loop_Bound:prog.exe:main.c:main@20->ave_adc:9-12:10
Loop_Bound:prog.exe:main.c:main@20->ave_adc:21-:15
Wcet_Call:prog.exe:main.c:main@20->ave_adc:5-15:737
Wcet_Call:prog.exe:main.c:main@20->ave_adc:5-15:1032
Wcet:prog.exe:main.c:main:18-21:1799
```

The final line shows that one execution of main, including all the functions it calls, needs at most 1799 machine cycles.

Analysis for stack usage

The Bound-T command to find an upper bound on the stack usage of the main function is:

```
boundt_8051 -no_time -stack prog.exe main
```

The output, again when the program is compiled with SDCC 2.7.0, is:

```
Device:prog.exe::::8051
Time_Unit:prog.exe::::Machine cycles.
Compiler:prog.exe::::SDCC
Stack:prog.exe:main.c:ave_adc:5-15:sp:0
Stack:prog.exe:main.c:main:18-21:sp:3
```

The final line shows that the main function and all the functions it calls need at most 3 octets of stack space in total. Note that this does *not* include the return address for main itself. Whether main has a return address depends on the start-up code which is not included in this analysis example.

3 WRITING ASSERTIONS

3.1 Introduction

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 explaining the 8051-specific aspects of assertions, in particular how to identify subprograms and variables by their machine addresses, how to specify the role performed by certain 8051 instructions, and which 8051-specific properties can be asserted for 8051 program parts.

3.2 Identifying subprograms by address

Bound-T for the 8051 uses the common format for 8051 code addresses. To identify a subprogram by means of its entry address write a string that gives the entry addresses in hexadecimal (base 16) form, using decimal numbers 0 - 9 and letters a, b, c, d, e and f (case-insensitive), followed by H or h to indicate hexadecimal. Subprogram addresses cannot be written in decimal (base 10) form.

The address can optionally be prefixed with "C:" or "c:" to ensure that the address is not confused with a symbolic identifier. For example, if the program contains a function named "abbah" you can write "c:abbah" or "C:ABBAh" to name the subprogram at address ABBA (hex) without risk of confusion with the function called "abbah". Another way to indicate that the string is an address and not an identifier is to add leading zeros, for example "0abbah".

The "C:" prefix or leading zeros may be useful for root subprogram addresses on the Bound-T command-line because the **address** keyword is not available there, unlike the case for subprograms named in assertions.

For example, here is an assertion that sets the execution time of the subprogram that begins at address 4AC hex (that is, 4AC is the address of the entry point of the subprogram):

```
subprogram address “C:4ACH”
  time 314 cycles;
end subprogram;
```

3.3 Code-address offsets

Some forms of assertions define code addresses by giving a *code offset* relative to a base address. For Bound-T/8051 a code offset is written as a decimal number, or as a hexadecimal number followed by the letter 'H' or 'h'. In both cases the offset is given in octet units and the number can be 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 4AC hexadecimal and the subprogram Abandon has the entry address 57B hexadecimal. The subprogram with the entry address 4D2 hexadecimal can then be identified in any of the following ways, among many others:

- Using the absolute address:

```
subprogram address "C:4D2H"
```

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

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

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

```
subprogram "Rerun" offset "38"
```

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

```
subprogram "Abandon" offset "-A9H"
```

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

3.4 Identifying variables by address

Spaces and sizes

The 8051 has a complex set of address spaces which leads to a fairly rich syntax of data addresses. We consider the Special Function Register (SFR) area as a data address space, although its “memory locations”, the SFRs, may not have normal memory semantics (reading an SFR may not return the last value written to that SFR, and reading or writing an SFR may have side effects such as the activation of peripheral input-output devices).

The smallest datum that can be addressed is a single bit; the largest is generally an octet (8 bits), although some special instructions like **ret** access 16-bit words. Compilers can of course define multi-octet variables such as 16-bit or 32-bit **int** or **long** variables in the C language. At present assertions in Bound-T/8051 are limited to octet variables in any octet-addressable memory space and bit variables in the bit-addressable memory space. Multi-octet variables cannot be used in assertions and are not used in the analysis, except for the registers **DPTR** and **PC**.

Register names and flag names

To identify a register in an assertion use the **address** keyword followed by a quoted string that names the register. Register names within the quoted string are case-insensitive.

All 8051 registers are located in the 8051 internal data address space or the SFR space and thus have a physical address in the range 0 .. 255 that can be used to read or write the register. Bound-T translates all register names to the corresponding internal data memory address. All 8051 flags (bit registers) are located in the bit-addressable SFR space and thus have a bit address in the range 128 .. 255 (decimal).

The name of a general 8-bit register consists of the letter R (or r) followed by the register number 0 – 7. For example, the string "r5" names register **r5**. Note that this syntax identifies a register *within the current register bank* as chosen by the *-reg_bank* option on the Bound-T command line. To name an "absolute" register (without going through a register bank) use the internal data memory address (see below).

The names of the **A** and **B** registers are "A" and "B". Bound-T translates these names to the corresponding Special Function Register (SFR) addresses.

The name of the carry flag **C** is "C". Bound-T translates this name to the corresponding bit address within the **PSW** octet in the SFR part of the bit-addressable internal memory.

The name of the **DPTR** is "DPTR". Its low and high octets can be named with "DPL" and "DPH" respectively.

Assertions are handled in the generic parts of Bound-T that do not know of the part-whole relationship between 16-bit registers such as **DPTR** and their octet parts, **DPL** and **DPH**. This means that an assertion on the value of **DPTR** does not imply any constraints on the values of **DPL** and **DPH**. Moreover, assertions on the values of **DPL** and/or **DPH** do not directly imply any constraints on the value of **DPTR**, but such constraints may arise indirectly at instructions that change **DPL** or **DPH** because Bound-T's model for such instructions recomputes **DPTR** by concatenating the values of **DPH** and **DPL**.

Numeric addresses

Variables located at a specific memory address are identified in assertions with the **address** keyword, followed by a quoted string of the form

"*M*:*num*" for decimal address *num*, or
"*M*:*num*H" for hexadecimal address *num*.

The symbol *M* stands for one letter that defines the memory space as shown in Table 12 below.

Table 12: Memory space symbols and address ranges

Symbol *M*	Memory space	Part	Address range (hex)
D	Internal data memory	directly addressed	0 .. 7Fh
		indirectly addressed	80h .. FFh
S	Special Function Register		80h .. FFh
B	Bit-addressable memory	within internal data memory	0 .. 7Fh
		within SFR space	80h .. FFh
X	External data memory		0h .. FFFFh
C	Code memory		0h .. FFFFh

If the string ends with “H” the address value *num* is interpreted as a hexadecimal number, otherwise as a decimal number.

Some examples:

`X:1000H` = external data address 4096 (decimal) = `X:4096`

`X:15000` = external data address 15000 (decimal) = `X:3A98H`

`D:1AH` = internal data address 26 (decimal) = register **r2** in bank 3.

`C:0200` = code address 200 (decimal)

The address string is case-insensitive, both for the memory space symbol and for the hexadecimal digits and the possibly trailing “H”.

Data in the stack

The 8051 architecture defines a “hardware” stack that is located in the internal data memory (often in the indirectly addressed upper half at addresses 80h .. 8Fh). Stack data are accessed via the SFR called **SP**, for Stack Pointer. The stack grows upwards – a **push** increases **SP** – and is “filled” in the sense that **SP** points to the last octet in use, not to the first unused octet. Programs can put local variables and parameters in the stack, which is useful for subprograms that must be reentrant. The call instructions push the return address onto the stack and the return instructions pop it.

Bound-T identifies data in the stack by an offset relative to the value of **SP** on entry to the subprogram under analysis. The actual value of the **SP** is generally not known to the analysis. The offset is positive for local variables and zero or negative for parameters. Offsets 0 and -1 refer to the high and low octets of the return address, respectively.

To name a stack variable in an assertion use the **address** keyword followed by a quoted string that consists of the letter V followed by the positive offset in decimal form, or in hexadecimal with a trailing letter H.

To name a stack parameter use the same form except start with the letter P instead of V.

Examples

Here are some assertions on the values of variables identified by their addresses or by register names:

```
variable address "R3" 0 .. 100;
-- The value of register R3 lies in the range 0 .. 100.

variable address "r3" 0 .. 100;
-- Same thing.

variable address "x:3fa7h" = 20;
-- The octet at external memory address hex 3FA7
-- (decimal 16295) has the value 20.

variable address “v1ah” < 15;
-- The stack local variable at offset +1a hex (= 26 decimal)
-- (relative to SP on entry) is less than 15.

variable address “p2” >= 32;
-- The stack parameter at offset -2 (relative to SP
-- on entry) is at least 32.
```

Data in code memory

Assertions on the values of data cells in code memory (with addresses of the form "C:*num*") are currently ineffective, because the constant-propagation part of Bound-T replaces all references to such cells by the constant value of the cell before the assertion is used. Moreover, since the value of the cell is known from the program code, asserting a different value would be a contra-factual assertion which could lead to incorrect analysis as explained in the Bound-T Reference Manual [2].

3.5 Time and space units

Assertions on the execution time of subprograms or calls (in the form **time** *t* **cycles**) use the unit of time (machine cycles or clock cycles) defined for the chosen device. See Table 1 (page 13) and Table 10 (page 20).

Assertions on stack usage (**stack usage** *u*) or final stack height (**stack final** *f*) use octet units.

3.6 Instruction roles

Some 8051 instructions can perform different or unusual roles in a program, depending on their context and operands. Bound-T needs to know the role, in order to model the instruction properly, and uses heuristic assumptions, sometimes supported by analysis, to choose the role for such multi-role instructions. If the automatically chosen role is not the best one, you can tell Bound-T which role to use by an assertion of the form

```
instruction ... performs a "role";
end instruction;
```

Table 13 below shows the *role* names understood by the 8051 version of Bound-T, and the instructions to which they can be applied.

Table 13: Instruction roles

Instruction	Role name	Role performed
jmp @A+DPTR	`"branch"`	A branch (jump) to an address computed dynamically by the sum of the **A** register (accumulator) and the **DPTR** register. Bound-T tries to find the possible target addresses by analysis.
	`"return"`	A return from the current subprogram to the calling subprogram. Bound-T does not try to analyse the possible values of the return address (**A+DPTR**).
	`"call"`	A call to an address (subprogram entry point) computed dynamically by the sum of the **A** register (accumulator) and the **DPTR** register, with return to the current subprogram. Bound-T does not try to analyse the possible values of the target address, but the possible callees can be asserted.
	`"tail call"`	Like `"call"` but the call returns to some higher-level subprogram, not to the current subprogram.

Instruction	Role name	Role performed
ret	`"branch"`	Either a return from the current subprogram (if Bound-T finds that the address popped from the stack is the return address), or a branch (jump) to whatever address is popped from the stack (otherwise). Bound-T tries to find the possible target addresses by analysis.
	`"call"`	A call to the address (subprogram entry point) that is popped from the stack, with return to the current subprogram. Bound-T does not try to analyse the possible values of the target address, but the possible callees can be asserted.
	`"tail call"`	Like `"call"` but the call returns to some higher-level subprogram, not to the current subprogram.

3.7 Properties

Assertions can give values or bounds for certain target-specific *properties* for specific subprograms or loops. The set of properties, and their meaning in the analysis, is entirely target-specific. Table 14 below describes the properties defined for the 8051 version of Bound-T.

Table 14: Assertable properties for the 8051

Property name		Meaning, values, and default value
returns	*Function*	Controls the analysis of **ret** instructions within specific subprograms. Overrides the command-line option *-returns* for this subprograms. Can be overridden by instruction role assertions for the **ret** instructions (see Section 3.6).
	Values	The value must be a single number (not an interval) in the range 0 .. 2. These values have the following meanings: 0 Use the command-line *-returns* option for this subprogram. 1 Analyse this subprogram as under *-returns=static* : assume that any **ret** instruction causes a return from the subprogram. 2 Analyse this subprogram as under *-returns=dynamic* : analyse each **ret** as a dynamic transfer of control that may resolve into a return or a jump.
	Default	Zero, which means to follow the command-line *-returns* option.

Consider, for example, the assertion:

```
subprogram “skip”
  property “returns” = 2;
end “skip”;
```

This assertion sets the “returns” property for the subprogram skip to 2, which means that Bound-T analyses each **ret** instruction within skip as a dynamic transfer of control, whatever the command-line option *-returns* may say or have as default.

4 THE 8051 AND TIMING ANALYSIS

4.1 Introduction

This chapter starts the reference-manual part of this Application Note by giving a compact overview of the 8051 processor architecture. This defines terms and concepts that later chapters use to describe how Bound-T models and analyses 8051 programs and in particular which 8051 features are fully supported and which are not.

4.2 The 8051 processor architecture

The 8051 [6] is an 8-bit microcontroller core. Reference [7] is a good introduction to the architecture of the processor and suitable programming methods with focus on programming in the C language using the C51 compiler from Keil [11, 12].

The 8051 has a "Harvard" architecture with separate program and data address spaces. Instructions can be 8, 16 or 24 bits wide. Data can also be read from the program memory. All memory is addressed by octet. Some on-chip memory octets can also be addressed by bit. Load and store instructions operate on 1-bit or 8-bit quantities only; to load or store multi-octet values as many load or store instructions must be used.

Data memory is divided into *internal* and *external* memory.

Internal data memory

The internal data memory is always on the same chip as the processor core and is architecturally limited to an 8-bit address (256 octets of data) but half of the address space is usually overlaid so that direct and indirect accesses address different hardware components. Some implementations provide further "paging" or "banking" of the internal data addresses.

The internal memory address space is divided into three ranges as follows:

- The range 0 to 127 (00 to 7F hex) can be addressed directly or indirectly. The first 32 locations (0 to 31, or 00 to 1F in hex) contain the 32 general-purpose registers (4 banks of 8 registers) on which more below. Thus, the registers can also be accessed as memory locations, directly or indirectly. The rest of this range (32 to 127, or 20 to FF in hex) can be used to hold the stack or application data.
- The range 128 to 255 (80 to FF hex) when addressed *directly* gives access to a number of *Special Function Registers* (SFRs). The set of SFRs depends on the implementation, but some are standard, for example the **A** and **B** registers, the **PSW**, the **DPTR** and the **SP** can all be accessed as SFRs. The 16-bit register **DPTR** is accessed as the two 8-bit SFRs **DPL** and **DPH**. More on these registers below.
- The same range, 128 to 255 (80 to FF hex) when addressed *indirectly* accesses a general-purpose internal data memory, often used to hold the stack and application data. Some implementations do not provided this extended internal memory.

Some parts of the internal data memory can be addressed and accessed by bit as well as by octet. The Program Status Word (**PSW**) is within such a part which means that each **PSW** status and flag bit can be accessed separately. The same holds for the **A** and **B** registers.

External data memory

The external memory uses a separate 16-bit address space. The external memory is usually off-chip but some current devices have some on-chip "external" memory to increase the total on-chip memory beyond the address limits of the internal data memory.

All accesses to the external data memory are addressed indirectly, either with the 16-bit Data Pointer register (**DPTR**) or by combining a pre-set "page" address (giving the high 8 bits of the address) with an 8-bit page-offset taken from an 8-bit register (**r0** or **r1**).

Program memory

The program code is located in a separate address space per the Harvard principle. Code is addressed by octet. In the basic 8051 the code address is 16 bits allowing a maximum of 64 Ko of code. Several implementations extend this by some form of "code banking" scheme. At present Bound-T does not support code banking.

There is a specific instruction to load data from the program memory, using an indirect address computed as **A**+**PC** or **A**+**DPTR**. There are no standard instructions for writing data or code to the program memory. Some implementations of the 8051 may be able to "self-program" in this way. Bound-T assumes that the program memory is not altered during execution.

Arithmetic

All arithmetic integer operations on 8-bit operands are supported in hardware, including multiplication and division. The only 16-bit operations supported in hardware are incrementation of the **DPTR** register and addition of an 8-bit offset to the **DPTR** as part of the "**@A+DPTR**" addressing mode. Floating point operations are not supported at all in hardware. No standard floating point type is defined.

Some devices that implement the 8051 core have additional arithmetic units as "peripherals" that are accessed as SFRs. The operands are loaded into specific operand SFRs and the result can be read from specific result SFRs. Bound-T does not model such arithmetic – it is "opaque" to the analysis.

Registers

There is a main accumulator register (**A**) and a secondary accumulator register (**B**), both with 8 bits. The instruction set uses eight general-purpose 8-bit registers, **r0** through **r7**. However, there are four banks of **r** registers, for a total of 32 general-purpose 8-bit registers. The bank in use is selected by a 2-bit field (**RS0**, **RS1**) in the Program Status Word (**PSW**). Registers can be addressed relative to the current bank (relative register number 0 .. 7) or using absolute register addressing (absolute register number 0 .. 31). The registers in fact occupy the first 32 octets of the internal data address space. Thus, they can also be accessed using direct or indirect 8-bit addresses.

Registers **r0** and **r1** can be used as page offsets in paged access to the external data memory.

An on-chip stack in the internal data memory contains the return addresses from subroutines and data pushed by **push** instructions. Since the internal memory is at most 256 bytes, and includes the banked registers, the stack must be less than 256 bytes. The Stack Pointer (**SP**) register is consequently only 8 bits wide.

The 16-bit Program Counter **PC** points to the next instruction in the program memory. It can only be accessed (implicitly) in the control-flow instructions (jump, call, return).

The 16-bit Data Pointer register (**DPTR**) acts mainly as a pointer to the external memory or to data in the program memory. Some extensions of the 8051 architecture have two or more **DPTR** registers with various means for selecting the “active” register. Two **DPTR** registers are useful for copying data from one place in memory to another because one **DPTR** can hold the source address while the other holds the destination address. Ate present Bound-T supports only one **DPTR**.

The Program Status Word (**PSW**) contains the condition flags for carry (**C**) and overflow (**OV**). There is also an auxiliary carry flag (**AC**) for BCD arithmetic, a parity flag (**P**) that shows the parity (number of '1' bits) of the value in the accumulator register **A**, and other flags that are not interpreted by Bound-T.

All of these registers are typically also accessible as Special Function Registers by direct addressing of the upper half of the internal data address space.

4.3 Static execution-time analysis of 8051 programs

The 8051 architecture is very suitable for static analysis by Bound-T. Instruction timing in no case depends on the data being processed. However, for some 8051 implementations the time may depend on the memory area that is accessed. Some devices may also have “accelerator” hardware such as various forms of caches. Bound-T generally assumes the worst case (for example, a cache miss) for such hardware.

In devices that have additional arithmetic or functional units in the SFR area the execution time of these additional functions may be variable; completion of a function may be indicated by a status bit in an SFR or by an interrupt. Bound-T does not itself model the execution time of such functions. For example, if completion is indicated by a status bit that is polled in a waiting loop you must assert the number of repetitions of this loop.

As there are almost no instructions for 16-bit arithmetic the automatic analysis of loop counters is currently limited to unsigned 8-bit computation (see section 5.4). Future versions of Bound-T for the 8051 may not have this limitation.

5 SUPPORTED 8051 FEATURES

5.1 Overview

This section specifies which 8051 instructions, registers and status flags are supported 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 8051, Bound-T also has generic limitations as described in the Bound-T User Guide [1]. For reference, these are briefly listed in section 5.1.

General support level

In general, when Bound-T is analysing a target program for the 8051, it can decode and correctly time all instructions, with minor approximations.

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 8-bit additions and subtractions assuming unsigned variables and no overflow or underflow. Bound-T correctly detects when this integer computation is overridden by other computations, such as multiplications in the same registers. Note that there are generic limitations on the analysis of pointers to variables.

Furthermore, because all registers (except the data pointer) are 8 bits wide and all arithmetic operations are performed with 8-bit entities, the processing of a wider variable requires several arithmetic operations to several registers or memory locations, chained via the carry flag **C**. Currently Bound-T does not understand that these operations actually process a multi-octet variable and cannot find and bound loop counters that are wider than 8-bit variables. In terms of the C language loop counters should be of type “unsigned char”.

Loops with signed counters or 16-bit or larger counters can be bounded only by user-given assertions. However, nominally signed 8-bit counters may be used if they stay within the range 0 .. 127 where the sign bit is zero, although the success of the analysis does depend on the kind of code that the compiler generates for such loops.

Reminder of generic limitations

To help the reader understand which limitations are specific to the 8051 architecture, the following compact list of the generic limitations of Bound-T is presented.

Table 15: Generic limitations of Bound-T

Generic Limitation	Remarks for 8051 target
Understands only integer operations in loop-counter computations.	No implications specific to the standard 8051. In devices that have additional arithmetic units as peripherals the results of such arithmetic are opaque.
Understands only addition, subtraction and multiplication by constants, in loop-counter computations.	No implications specific to the 8051.
Assumes that loop-counter computations never suffer overflow or underflow.	No implications specific to the 8051. However, note that overflow or underflow is not unlikely in 8-bit computation, and may even be used deliberately. Loop-bounds analysis succeeds only for loops that repeat less than 256 times. For loops that use the **djnz** instruction the analysis is correct only if the initial value of the counter is positive (not zero) because a zero initial value would lead to underflow on the first execution of the **djnz**.
Can bound only counter-based loops.	No implications specific to the 8051.
May not resolve aliasing in dynamic memory addressing.	Since the general registers in the 8051 are addressable as internal data locations 0 .. 31, unresolved aliasing may affect the analysis of the values of the registers **r0** through **r7**. In most larger processors registers cannot be addressed indirectly and are therefore protected from aliasing.
May ascribe the wrong sign to an immediate (literal) constant operand.	No implications specific to the 8051.

5.2 Main assumptions

Bound-T for the 8051 makes the following 8051-specific assumptions about the target program under analysis:

- The register bank is not changed within the code that is included in one analysis. The command-line option *-reg_bank=b* defines the bank in use. Bound-T does not try to track changes in the register-bank selection bits (**RS0**, **RS1**) in the **PSW**.
- The base address of paged addressing is not changed within the code that is included in one analysis. The command-line option *-page=address* defines the page-base address.
- The code memory is read-only. If the program reads data from to code memory using a **movc** instruction, and Bound-T can resolve the address that is read, and the executable file under analysis loads a value into this address, this value is returned by the **movc**.

5.3 Instructions and computations

Bound-T for the 8051 models the main computational effect of most 8051 instructions accurately, within the generic limitations of Bound-T and within the current 8051-specific limitation to 8-bit computations (except for the **PC** and **DPTR**). This section describes the computational effects that are modelled approximately or not at all.

Registers and memory

Most registers and memory locations in the 8051 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.
- All memory locations and SFRs are currently assumed to have standard non-volatile memory semantics, that is, reading the location returns the last-written value.

In reality some SFRs may not behave in this non-volatile way. For example, reading an SFR that represents a bidirectional 8-bit port may yield the states of the input lines, not the value that was last written to the SFR to set the states of the output lines. Future versions of Bound-T will provide means to define some SFRs as "volatile" which will mean that the value read from the SFR can differ from the last-written value.

Bit-addressed internal data memory

Each bit-addressable bit in the internal data memory is modelled as a cell. By default Bound-T models the interaction between these bit cells and the "host" octet cells that contain the bits as follows:

- An instruction that assigns some value to a bit cell also has the effect of assigning an opaque value to the host octet cell. However, if the value assigned to the bit cell is known, it sets a range constraint on the host octet cell. For example, assigning 1 to bit 6 means that the (unsigned) value of the host octet is at least $2^6 = 64$.
- An instruction that assigns some value to a host octet cell also has the effect of assigning opaque values to the bit cells within this host. However, if the value assigned to the host is known, each bit cell is assigned the corresponding bit from that value. For example, if the host octet is assigned the value 17 then bits 0 and 4 within the host are set (assigned the value 1) and the other bits in the host are cleared (assigned the value 0).

This interaction betwen bit cells and their host octet cells can be disabled with the command-line option *-no_bit_mix*. This can reduce analysis time in some cases but can lead to wrong analysis results if the bit/octet interaction is significant for program flow.

External data memory

Each octet in the external data memory is modelled as a cell. Since all accesses to the external data memory are indirect, Bound-T must always try to analyse the computation of the address (in **DPTR**, **r0**, or **r1**) to find out which cell is accessed. This analysis often succeeds easily (by constant propagation) because even accesses to statically known variables in external memory use the indirect mechanism. Thus, an access to external data via **DPTR** is often immediately preceded by instructions that set **DPTR** to a statically known value (the address).

For paged accesses, where only the low 8 bits of the address are computed (into **r0** or **r1**) Bound-T assumes that the high 8 address bits have the constant value set by the *-page* option.

Data from the program memory

The **movc** instruction reads an octet from the program memory. It is often used to access constant data embedded in the program code. It always uses an indirect, dynamic address, which is either **A**+**DPTR** or **A**+**PC**. Bound-T uses constant propagation and arithmetic analysis to try to resolve the actual address. If this succeeds Bound-T fetches the octet value from the program's memory image, making the value statically known for further constant propagation or arithmetic analysis. This often happens in the partial evaluation of switch handlers, as explained in chapter 7.

Computations

For most instructions that do some arithmetic computation with a non-constant result, the following results are modelled as opaque values:

- the individual bits within the target register;
- the parity flag **P**, when the accumulator **A** is the target register;
- the **PSW** as a whole, when the instruction changes any bit in the **PSW**;
- the host octet containing the target bit, for instructions that assign to a bit cell (however, the value of the host octet can be bounded to a range).

Table 16 below lists further instruction-specific limitations of the model .

Table 16: Instructions modelled incompletely

Instruction	Model deficiencies
inc DPTR	The **DPL** and **DPH** registers become opaque.
mov @r0, .. **mov @r1, ..**	If the target is a bit-addressable octet, the effect on the bit cells within that octet is not modelled (the bits are not even made opaque).
push	The instruction has no effect on any one-bit cells. In other words, we assume that **SP** does not point into bit-addressable memory, or at least that the effect of **push** on bit locations is unimportant.
rl A, rr A	Result is opaque in **A**.
rlc A, **rrc A**	Result is opaque in **A** and **C**.
da A	Result is opaque in **A** and **C**.
swap A	Result is opaque in **A**.
xchd	Both operands become opaque.

5.4 Some consequences of the limited arithmetic model

Arithmetic model without underflow and overflow

Bound-T for the 8051 at present uses a simplified model of the arithmetic computation in which overflow and underflow are ignored, for the most part. For example, the model of the **A** register is an arithmetic cell (an integer variable) that notionally can have any integer value. This model agrees with reality only when the variable value is in the range 0 .. 255. If the program under analysis causes the **A** register to overflow or underflow Bound-T may give wrong results. Consider, for example, the following instruction sequence:

```
clr A
dec A
cjne A, #255, label
```

The first instruction sets **A** to zero and the second instruction decrements it. In the real processor the decrementation causes underflow and stores the value 255 in **A**, thus the third instruction (compare **A** to 255 and jump if not equal) *never* jumps to the **label**. In the current

Bound-T arithmetic model, however, the decrementation stores the value -1 in the cell that models **A**, thus the analysis wrongly concludes that the third instruction *always* jumps to the **label**.

Tidorum is working on extending the Bound-T arithmetic model to include overflow and underflow and expects to provide such a model in some future version of Bound-T for the 8051 and other processors.

The present approximate arithmetic model has several consequences that should be considered when using Bound-T on 8051 programs. If you are writing 8051 programs that will be analysed by Bound-T you may find it useful to select coding styles that match the present arithmetic model.

Unsigned interpretation of literal operands

Bound-T models all literal (immediate, constant) operands in 8051 instructions as unsigned, non-negative numbers. Instructions that use negative literal values (by two's complement) are not modelled accurately. For example, in the real processor the instruction **add A, #254** decreases the value of **A** by 2 (because 254 = –2 in 8-bit two's complement) and thus has the same effect as the instruction pair **clr C; subb A, #2**. However, in Bound-T's model only the latter instruction pair has this effect, while **add A, #254** always seems to increase the value of **A**.

5.5 Time accuracy and approximations

Bound-T reports WCET values that take into account most of the timing features of the 8051. However, certain 8051 devices and systems may have timing properties that are not modelled or for which a pessimistic (worst-case) model is used. See Table 1 in section 2.2 (page 13).

6 SUBPROGRAM CALLS AND PARAMETERS

6.1 Subprogram calls in the 8051

In this chapter, we discuss how 8051 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.

Static calls

The 8051 instruction set contains instructions specifically intended for subprogram calls: **acall** and **lcall**. Both instructions work in the same way; they differ only by the encoding and range of the target address – the entry address of the callee – so we will use the symbol **call** to refer to them both.

When the processor executes a **call** it first pushes the return address on the stack (in the internal data memory, referenced by the **SP** register) and then transfers control to the callee at the address encoded in **call** instruction. The callee finishes by executing the **ret** instruction which pops the return address from the stack into the program counter **PC**. so that execution continues in the calling subprogram with the next instruction after the **call** instruction.

The callee address in a **call** is always statically encoded as part of the instruction. Thus Bound-T always knows the callee address and can build the call-graph statically.

Dynamic calls and function pointers

The 8051 has no *single* instruction that calls a subprogram at a dynamically computed address. Instruction *sequences* with this effect of course exist. One such sequence is to put the computed callee address in the **DPTR** register and **call** a "helper" subprogram that contains two instructions: **clr A** followed by **jmp @A+DPTR**.

In the C programming language dynamic calls correspond to calls that use *function pointers* instead of static function names. Different C compilers may generate different sequences of 8051 instructions to implement calls through function pointers.

At present Bound-T for the 8051 does not recognise any such instruction sequences as dynamic calls. In the example sequence, the instruction **jmp @A+DPTR** would be analysed as an unresolved dynamic jump and the analysis of the caller would not include the analysis of the callees. However, you may assert that the instruction performs the role of a call, which makes Bound-T model the instruction as a dynamic call, as described in Section 3.6. Even so, Bound-T is usually unable to resolve the call (find the possible callees) by itself, so you must usually also assert the possible callees for each dynamic call.

If your program uses dynamic calls heavily, please do consult with Tidorum; perhaps Tidorum can extend Bound-T to recognise the dynamic calls that your C compiler generates. This would remove the need for instruction role assertions and still let you use assertions to list the possible callees of each dynamic call. Automatic analysis to find the callees is not possible in Bound-T in its current form.

6.2 Parameter passing

Statically allocated parameters and locals

The 8051 instruction set and memory architecture make it cumbersome to access data on a stack. This holds both for the "hardware" stack (accessed via the **SP** register) and for "software" stacks that may be defined by a compiler or by programming convention. It is therefore common to pass parameters to 8051 subprogram through statically allocated memory locations in the internal or external data memory. Of course such a subprogram is not reentrant and cannot be recursive (except, perhaps, tail-recursive) but many 8051 applications need neither reentrancy nor recursion.

Local variables are often handled in the same way, as statically allocated data.

A simple way to manage statically allocated parameters and local variables is to assign separate memory locations for each separate parameter and local variable in the program. This can be done subprogram per subprogram. Advanced compilers for the 8051 can use the program-wide call-graph to determine which subprograms can be active at the same time (in the same sequence of nested calls) and which cannot. Subprograms that can never be active at the same time can use the same static memory locations for their parameters and local variables. Such "overlaying" reduces the total amount of memory used.

Bound-T can analyse parameter-passing through statically allocated locations, whether or not the compiler lets several subprograms share the same locations. For the 8051 the analysis of parameters, and the resulting context-dependent analysis of the callee, is limited to 1-bit or 8-bit data.

Register parameters

Some parameters may be passed via registers (**r0** .. **r7**, **A**, **B**, **C**, **DPTR**). For Bound-T such parameters are similar to statically allocated memory locations (the register number is statically allocated). Analysed parameter values are limited to 1-bit or 8-bit data.

Stack-allocated parameters and local variables

Subprograms on the 8051 can be made reentrant by passing parameters in some kind of stack or in registers. Local variables for reentrant subprograms must also be allocated in a stack or to registers. Compilers commonly use the native 8051 stack (the **SP** stack) primarily for return addresses and define a separate "software" stack for parameters and local data. The native stack may also be used to save and restore data, for example callee-save registers that will be used in a subprogram.

Bound-T models data cells on the **SP** stack when they are accessed with **push** and **pop** instructions. The 8051 has no **SP**-based indexed addressing and no standard frame pointer register. Thus a compiler that places parameters or local variables in the **SP** stack will use some compiler-specific method to access those data (probably using indirect addressing with the **r0** or **r1** registers). Bound-T uses its constant-propagation analysis to discover and model such accesses to the **SP** stack by offsets from the initial value of **SP** on entry to the subprogram.

At present Bound-T has non specific models for software stacks. An access to data on a software stack will probably appear to Bound-T as an unresolved dynamic (indirect) memory access that yields an opaque value. While Bound-T can analyse programs that use software stacks it cannot use data in such stacks for the analysis of loop bounds, nor can it find upper bounds on the space-usage of software stacks.

Registers and memory locations modified by a call

Bound-T initially assumes that any call can modify any register. In other words, Bound-T assumes that there are no “callee-save” registers. When Bound-T analyses the callee it finds the actual set of modified registers (assuming that no register is modified indirectly by a pointer). The actual modified-register set enters the final analysis of the caller.

For calls to subprograms that are not analysed (because their resource bounds have been asserted) the inital assumption, that the call can modify any registers, is retained in the analysis of the caller, but it can be modified by invariance assertions.

Bound-T assumes that a callee subprogram does not modify any location in the **SP** stack that was pushed by the caller subprogram, or any higher-level subprogram. That is, Bound-T assumes that a subprogram modifies only its “own” part of the **SP** stack.

7 COMPILER SUPPORT

7.1 Important compiler features

Different cross-compilers for the 8051 are likely to generate different code for the same source program. Some differences in the code are unimportant for analysis by Bound-T although they can influence the results of the analysis, for example the execution-time bounds. Other compiler-specific properties, idioms or conventions in the code can greatly help or hinder the analysis. Bound-T is sensitive to the following properties of the program under analysis, and these properties are (partly) defined by the compiler:

- *Calls*: The compiler decides what code to generate for subprogram calls and returns. In most cases the standard 8051 call and return instructions (**lcall**, **acall**, **ret**, **reti**) will be used, but some compilers may use different instructions in some cases, for example to implement code banking (not supported in Bound-T at present).
- *Parameter* passing: The compiler decides how to pass parameters between the caller and the callee. Since the 8051 has weak stack instructions parameters are often placed in statically allocated storage, but for recursive or reentrant subprograms the compiler must use some kind of stack. At present Bound-T supports only the native **SP** stack for this purpose.
- *Local* variables: In principle the compiler decides how to allocate memory for local variables (guided by the C keywords *static*, *auto*, and *register*). Many compilers for the 8051 support additional keywords or pragmas that give the programmer more control over this allocation, for example to choose between internal or external data memory. As for parameters, local variables are often placed in statically allocated storage. Bound-T can analyse statically allocated variables and local variables in the **SP** stack.
- *Register banks*: In principle the compiler chooses the register bank to be used at each point in the code, but most 8051 compilers support additional keywords that let the program specify the register bank, usually on a subprogram level. Bound-T assumes that the same register bank is used throughout the part of the program that is included in one analysis, that is, in all subprograms in the call-closure of the root subprogram.
- *Generic pointers*: The 8051 uses different instructions for indirect access to data in the internal memory, in the external memory, and in the code. Thus a C pointer cannot, in general, be implemented as a simple address, but it must also indicate which of these address spaces is meant. Such pointers are commonly called “generic” pointers. C compilers for the 8051 often use library routines to create and use generic pointers, which may complicate static analysis of the program. However, the compilers usually also provide keywords to define special pointer types, for example a pointer that can only point to the internal data memory and can thus be implemented simply and efficiently as an 8-bit address.
- *Name mangling*: Compilers often make some systematic changes to the source-level (C-level) identifiers of subprograms and variables when the compiler creates the corresponding linker symbols. All inputs to and outputs from Bound-T use the linker symbols. For example, if you need to write assertions to guide the analysis of a subprogram the assertion must use the linker symbol to identify the subprogram.

For the above properties Bound-T has the same abilities and limitations for all compilers. The following properties, however, require some compiler-specific support or adaptation in the analysis:

- *Program file format*: A given compiler/linker can usually generate (store) the executable program in one or a few different forms or file formats such as AOMF or Intel-Hex. Bound-T for the 8051 can read several formats as explained in section 2.1. The most important difference between the formats is in the amount and detail of the source-code symbolic (debugging) information that comes with the raw machine code.
- *Switch-case* statements: The switch-case is the most flexible and variable control structure in the C language, and this flexiblity is reflected in the complexity of the code that is generated. For example, the data-type of the switch expression can have a large effect on the code, and the code generated for a densely numbered sequence of cases can be quite different from the code for sparsely numbered cases. The dense case is likely to use some form of dynamic jump via a table of jumps or addresses, while the sparse case often uses compiler-specific “helper” routines that have unusual calling sequences.
- *Peculiar calling sequences*: Some compilers use library routines with peculiar, non-standard calling sequences. In one common aberration the call instruction is followed by constant parameter data embedded in the code, for example a table that represents a sparse switch-case structure. If such a call is analysed in the normal way this data would be interpreted as code, with results that may be amusing but surely useless. Bound-T detects some such peculiar routines and analyses them in special ways.

The rest of this chapter explains the compilers that Bound-T supports, explaining the properties of the code that the compiler generates, how Bound-T analyses that code, and which compiler options and features are supported or not supported. However, the information may be incomplete, in particular concerning library routines with peculiar calling sequences. When we say that Bound-T "supports" a compiler we mean that Bound-T has some knowledge of that compiler. It does not mean that Bound-T can analyse all programs compiled by that compiler.

7.2 IAR C compiler

Introduction

IAR Systems [9] supplies a C, C++ and EC++ compiler for the 8051. The rest of this section is based on the Reference Guide [10] and our own experiments and tests.

C or C++

While Bound-T may be able to analyse some parts of a C++ or EC++ program compiled by the 8051 IAR C/C++, it has no specific support for C++ features such as virtual function calls. Virtual function calls will probably appear as unresolved dynamic jumps in the analysis.

Those features of C++ or EC++ that do not cause more dynamic control-flow or dynamic data accesses should not cause analysis problems.

Program formats

The 8051 IAR C/C++ compiler (with the XLINK linker) can generate executable programs in several formats, but the most complete format is IAR's own Universal Binary Relocatable Format, UBROF. Bound-T can read 8051 programs in UBROF form (although Bound-T does not yet use *all* of the UBROF information).

Register banks

The 8051 IAR C/C++ run-time system and all non-interrupt application subprograms use a common register bank – the default register bank – that is usually bank 0 but can be set to some other bank by a linker command. The compiler supports a register_bank pragma by which an interrupt handler function can use a different register bank. In this case the compiler generates code to switch register banks but does not save or restore the registers in the chosen bank.

Thus the 8051 IAR C/C++ compiler is compatible with Bound-T as regards register banks. The default option in Bound-T is *-reg_bank=0* but it can be set to another value when an interrupt handler is analysed.

Paged memory

The 8051 IAR C/C++ compiler assumes that paged data (*pdata*) uses one fixed page of external data memory. The page is chosen by linker commands. This is compatible with Bound-T as long as the Bound-T option *-page* is set to the same value.

Subprogram call and return

For non-banked code the 8051 IAR C/C++ compiler uses the ordinary call and return instructions, agreeing with Bound-T. The question of function pointers is not yet explored.

Calling conventions and parameter passing

The 8051 IAR C/C++ compiler provides a choice of six different calling conventions to control the (default) allocation and placement of parameters and local variables.

Two of the IAR conventions use statically allocated storage and do not support reentrant subprograms. The remaining four conventions use various types of stacks and do support reentrancy.

At present Bound-T for the 8051 can model only computations using statically allocated parameters, corresponding to the two static IAR conventions: “data overlay” and “idata overlay”. These conventions use the internal data memory and therefore have tight limits on the total size of parameters and local variables. The IAR compiler does not provide a convention that uses statically allocated external data memory. The programmer can use the xdata keyword to place local variables in external memory, but not parameters.

The four stack-based IAR calling conventions are discussed below, under “Stacks”.

The IAR compiler defines registers **r6** and **r7** as “permanent” or callee-save registers. The **DPTR** register is also callee-save for some kinds of subprograms. At present Bound-T does not treat **r6** and **r7** specially. Bound-T initially assumes that any call may alter any register; analysis of the callee may then reduce the set of altered registers and should thus reveal that **r6** and **r7** (and perhaps **DPTR**) are not altered. However, if you prevent the analysis of the callee by asserting its resource bounds it may help the analysis of the callers if you also assert that **r6** and **r7** and perhaps **DPTR** are invariant in all calls of this callee.

Stacks

The 8051 IAR C/C++ compiler provides four reentrant calling conventions, respectively using the 8051 processor stack in the internal memory, a software-managed stack in paged external memory, a software-managed stack in non-paged external memory, or the “extended” hardware-managed external-memory stack that is available in some 8051 devices. As explained above in section 7.1 Bound-T is unable to analyse computations that use data on the

stack. Bound-T can analyse programs that use the first three reeentrant calling conventions but will not find loop-bounds that depend on stack-allocated parameters or local variables. Bound-T does not support the “extended stack” option.

Switch-case statements

Under investigation. Information to be provided in a later issue.

Library subprograms that violate the calling standard

Under investigation. Information to be provided in a later issue.

Compiler options

Table 17 below lists those IAR compiler options that influence the generated code and explains how Bound-T supports, or does not support, each option. Options not listed are supported. The options are shown in their long form. Please refer to the IAR manual [10] for the corresponding short forms.

If you have a pressing need to use an option that Bound-T does *not* support now, please contact Tidorum to discuss your needs.

Table 17: IAR compiler options

Option	Value	Supported?	Remarks
--calling_convention=			
	data_overlay	Yes	
	idata_overlay	TBC	
	idata_reentrant pdata_reentrant xdata_reentrant	Yes/**No**	These options make the compiler place all parameters and local variables in a stack. Bound-T cannot analyse computations using stacked data; thus it will not find any loop-bounds that depend on stacked data. You can work around this with assertions, of course.
	ext_stack_reentrant	**No**	Bound-T supports only the standard 8-bit **SP** register, no extension registers.
--char_is_signed		Yes/**No**	Bound-T models all literal operands as unsigned (non-negative) numbers. Analysis of computations involving negative literals will probably fail.
--code_model=			
	tiny near	Yes	
	banked far	**No**	Bound-T does not support banked code, nor extended code memories larger than 64 Ko.
--core=			
	plain	Yes	
	tiny	TBD	To be investigated. These devices are rare, it seems.
	extended	**No**	Bound-T does not support these extensions.
--data_model=			

Option	Value	Supported?	Remarks
	tiny small large	Yes	
	generic	Yes/**No**	Bound-T has no model for “generic” pointers; it is unable to resolve even statically set pointers of this kind.
	far	**No**	Bound-T does not support data space beyond 64 Ko.
--debug		Yes, and recommended	Bound-T needs the debugging information to interpret assertions and parameters and to annotate the analysis results with source-code identifiers and locations.
--dlib_config		Yes, TBC	Tidorum has not analysed all the library routines for all library variants.
--dptr=			
	16,1	Yes	Bound-T supports one **DPTR** register of 16 bits.
	any other value	**No**	At present Bound-T does not support multiple **DPTR** registers or **DPTR** registers of more than 16 bits.
--ec++ --eec++		Yes/**No**	Bound-T cannot analyse virtual function calls so they appear as unresolved dynamic jumps. In other respects the analysis of C++ code is the same as for C code.
--extended_stack		**No**	Bound-T supports only the standard 8-bit **SP** register, no extension registers.
--nr_virtual_regs=*n*		Yes TBC	
--omit_types		**No**/Yes	This option makes the compiler omit the information about the types of variables. This may make Bound-T omit these variables from its own symbol-table in which case assertions cannot use these variables (by name).
--place_constants=			
	data data_rom	Yes, but...	At run time the constants reside in writable data memory, therefore Bound-T does not consider the values to be static constants, but (probably) opaque.
	code	Yes	Bound-T understands that data embedded in the code memory is read only and finds the constant value from the program file.
-s -z		Yes TBC	Tidorum has not tested all sorts of C code under all optimization levels.

7.3 Keil C compiler

Introduction

The C51 compiler from Keil [11] is one of the best known C compilers for the 8051. The rest of this section is based on the Keil C51 User's Guide [12] and our own experiments and tests. A variant called CX51 supports extended versions of the 8051 that provide up to 16 Mo of code space. Bound-T does not support such devices so this section talks only of the C51 compiler.

Keil provides two linkers: BL51 and LX51. Their differences are usually not important for Bound-T.

Program formats

Keil C51 can generate executable programs in AOMF form as defined by Intel [15] with symbolic debugging information extensions defined by Keil [16]. Reference [18] explains how Bound-T supports this AOMF format. Keil C51 and in particular the LX51 linker can optionally generate programs in the newer Keil-defined OMF2 format, which Bound-T can also read.

Register banks

By default Keil C51 compiles code to use register bank 0, agreeing with the default bank in Bound-T. The REGISTERBANK directive can used to define another register bank. This directive defines a "common" register bank in the sense that the caller and callee are expected to use the same register bank; the compiler does not insert bank-switching code.

Keil C51 supports the *using* keyword that specifies the register bank that a subprogram uses. The compiler also generates code to switch register banks on entry to and return from this subprogram. Subprograms that use different register banks can call one another. In contrast, Bound-T assumes that all subprograms in the same call-graph use the same register bank, set with the option *-reg_bank*. However, it may be able to analyse call-graphs in which the register bank is changed if there is no significant data-flow across the points of change.

Paged memory

In the Keil tools, the base address of the paged external memory is defined by the linker directive PDATA in BL51 or the equivalent directive (TBD) in LX51. There is no default value in the linker itself; there may be a default value in the Keil IDE. The Bound-T default *-page=0* corresponds to PDATA(0).

Subprogram call and return

By default Keil C51 uses the ordinary call and return instructions, agreeing with Bound-T. The question of function pointers is not yet explored.

For code banking or some extended 8051 devices C51 can use extended or unusual forms of call and return code. The compiler directives that lead to such code are marked as "not supported" in Table 18 and Table 19 below.

Parameter passing

By default Keil C51 passes subprogram parameters in registers and statically allocated memory locations (internal or external, depending on options and source-code keywords). This is compatible with Bound-T.

Stacks and reentrant subprograms

Keil C51 never uses the processor stack for parameters, only for return addresses. This is compatible with Bound-T.

For reentrant subprograms Keil C51 can use software-defined stacks in internal data memory (SMALL memory model), in paged external memory (COMPACT model) and in general external memory (LARGE model), while still using the processor stack for return addresses. Different subprograms in the same program may use different models so there may be up to four stacks in use at the same time, in the same program.

As explained above in section 7.1 Bound-T is unable to analyse computations that use data on the software-defined stacks, whatever the memory model.

Switch-case statements

In our experience Keil C51 can generate three kinds of code for switch-case statements, depending on the type of the index expression and on the numbering of the case branches:

- When there are few cases, C51 generates a cascade of in-line comparisons and conditional jumps, similar to the code that would result from the cascade of if-else if statements that is equivalent to the switch-case. All jumps have static targets. The code for the cases is embedded within the cascade of comparisons and conditional jumps.
- When there are more cases, but the cases are numbered densely (consecutively) and the index type is an 8-bit type, C51 generates a table of **ljmp** instructions, one for each case. The code for the switch-case statement uses the dynamic jump instruction **jmp @A+DPTR** to jump into the indexed point in this table, and then the **ljmp** at that point in the table jumps to the code for this case.
- In other cases (a considerable number of sparsely numbered cases, or a wider index type) C51 generates a *switch table* that contains the case numbers and the address of the code of each case. C51 puts this data table in the code memory, immediately after a call to a special *switch handler* routine in the C51 run-time library. The switch handler routine scans the switch table, matches the case numbers to the index value, and when they match, jumps to that case using a dynamic jump (**jmp @A+DPTR**, again).

Bound-T can of course analyse the first form (in-line comparisons and static jumps) without problems.

For the second form, using a dense table of jumps, Bound-T applies arithmetic analysis to the computation of the switch index – in particular to the comparisons that check that the index is in range – to find all the entries in the jump table, thus all statically possible target addresses of the dynamic jump, **jmp @A+DPTR**, into the jump table. At present this works only for 8-bit index types (C "char" types) because Bound-T for the 8051 models only 8-bit computations. Moreover, even for 8-bit types C51 sometimes uses the **MUL** instruction to compute the jump address, and then Bound-T cannot find the jump table (because **MUL** has a 16-bit result). If Bound-T finds the jump table it can analyse each jump table entry in the normal way because the **ljmp** instructions in the table have static targets.

For the third form, using a switch table and a switch handler routine, Bound-T partially evaluates the switch handler with respect to its constant parameter: the switch table. This method is explained in reference [23]. The partial evaluation of the dynamic jumps in the switch handler turns them into static jumps, at which point the partial evaluation ceases and normal analysis of the case branches continues. This method is insensitive to most internal details of the Keil C51 switch table formats and switch handler routines but relies on the following assumptions:

– The name of the switch handler is ?C?CCASE, ?C?ICASE, or ?C?LCASE, and this symbol is present in the debugging information in the program file.

– The switch handler is invoked by a call statement (**acall** or **lcall**) and the switch table is placed in the code immediately after the call statement (thus the return address points to the start of the table).

– The switch handler ends by executing a **jmp @A+DPTR** instruction to go to the chosen case branch.

Library subprograms that violate the calling standard

Under investigation. Information to be provided in a later issue. If you experience any problems with library subprograms please contact Tidorum.

Symbolic debugging information

Keil C51 makes some systematic changes ("mangling") when converting the C name of a subprogram to a linker symbol:

– If the subprogram passes some parameters in registers, C51 puts an underscore '_' in front of the name: "foo" becomes "_foo". An underscore is not added for subprograms that pass all parameters in fixed memory locations.

– Other cases under investigation. Information to be provided in a later issue.

Compiler options

Table 18 below lists those Keil C51 compiler directives (options and pragmas) that influence the generated code and explains how Bound-T supports, or does not support, each directive. Unless otherwise stated support for a directive *X* implies support for the negative form *NOX* of the directive, too, when the negative form exists. Directives not listed are supported. Table 19 below lists the linker directives for BL51 and LX51 in a similar way.

If you have a pressing need to use a directive that Bound-T does *not* support now, please contact Tidorum to discuss your needs.

Table 18: Keil C51 compiler directives

Directive	Supported?	Remarks
AREGS	Yes	Bound-T always models register accesses as "absolute" direct addresses (relying on the specified register bank in use). Thus the program can use relative (**r0** .. **r7**) or absolute (internal data address 0 .. 31) register addresses, as it pleases.
BROWSE	Yes	The format of AOMF "source browse" records is not described in [16], thus Bound-T cannot make use of them and will skip them while reading the AOMF program.
COMPACT	Yes	Assuming that the correct base address of the paged part of the external data memory is set using the Bound-T *-page* option.
DEBUG	Yes, and recommended	Bound-T needs the debugging information to interpret assertions and parameters and to annotate the analysis results with source-code identifiers and locations.
DISABLE	Yes	
LARGE	Yes	
MDU_F120 MDU_R515 MODDA	Yes TBC	These directives make use of additional computational units that are accessed as SFRs. Bound-T can analyse the code that reads and writes the SFRs but has no knowledge of the computations performed by these SFRs nor of how long these computations can take.

Directive	Supported?	Remarks
MOD517 MODA2 MODAB2 MODC2 MODDP2 MODH2 MODP2	**No**	These directives make C51 use extended sets of forms of Data Pointer Registers. Bound-T supports only one **DPTR** register. Please contact Tidorum if you need support for these extensions.
AMAKE	TBD	
OBJECTADVANCED	Yes	TBC for shared entry code (may need “integrated” decoding).
OBJECTEXTEND	Yes, and recommended	Increases the amount and quality of symbolic debugging information that Bound-T can use.
OMF2	Yes	Bound-T can read programs in the absolute subset of the OMF2 format.
ONEREGBANK	TBD	
REGISTERBANK	Yes	Assuming that the same register bank is set for Bound-T using the Bound-T option *-reg_bank* and is used for all subprograms in one analysis (in the analysed call-graph).
REGPARMS	Yes	Registers are statically addresses storage locations, so Bound-T can analyse parameters passed in registers.
RET_PSTK RET_XSTK	**No**	These directives make C51 generate nonstandard handling of return addresses.
ROM (SMALL) ROM (COMPACT) ROM (LARGE)	Yes	These directives make C51 choose short or long forms of jump and call instructions. Bound-T supports all forms.
ROM (D512K) ROM (D16M)	**No**	These directives make C51 use extended forms of jump and call instructions that Bound-T does not support.
SMALL	Yes	
STRING	Yes	The value specified for this directive (CODE, XDATA, or FAR) may affect the string-manipulation code, making it easier or harder to analyse, but not impossible.
VARBANKING	**No**	Bound-T does not support data-memory extension by “data banking”.
XCROM	Yes	

Table 19: Keil BL51/LX51 linker options

Option	BL51	LX51	Supported?	Remarks
BANKAREA	•	•	**No**	Bound-T does not support code banking.
BANKx IBANKING	•		**No**	Bound-T does not support code banking.
NOAJMP NOINDIRECTCALL NOJMPTAB	•	•	**No**	Bound-T does not support code banking.
OVERLAY	•	•	Yes	

Option	BL51	LX51	Supported?	Remarks
PDATA	•		Yes	Ensure that the value agrees with the value of the Bound-T *-page* option. The default in Bound-T is *-page=0*, corresponding to PDATA(0).

7.4 SDCC – Small Device C Compiler

Introduction

The *Small Device C Compiler* is a free, open-source C compiler [13] for some small target processors. It uses the *ASxxxx* assembler and the *ASLINK* linker [21]. The system supports the basic 8051 architecture and a few variants. The rest of this section is based on the SDCC User Guide [14] and our own experiments and tests.

Program formats

SDCC and ASLINK can generate executable programs in Intel Hex form or in AOMF with Keil extensions. References [19] and [18] explain how Bound-T supports these formats, respectively. The AOMF format contains debugging information, but not in very nice form; for one thing, all identifiers are converted to upper case. The Intel Hex format contains no debugging information, but Bound-T can instead read the debugging information from the "CDB" file that SDCC also generates. CDB files have the correct (original) identifiers. Therefore the combination Intel-Hex + CDB is recommended. Reference [22] explains how Bound-T supports CDB.

Register banks

SDCC uses register bank 0 (internal data locations 0 .. 7) except for functions that are explicitly marked with the *using* keyword to use some other bank. These are typically interrupt handler functions. Thus SDCC is compatible with Bound-T in this respect; the default option in Bound-T is *-reg_bank=0* but it can be set to another value when an interrupt handler is analysed.

Paged memory

By default SDCC places paged data (*pdata*) in the first page of external data memory. This is also the default in Bound-T (default option *-page=0*).

Subprogram call and return

SDCC uses the ordinary call and return instructions, agreeing with Bound-T. The question of function pointers is not yet explored.

SDCC often performs *tail call optimization*: when the last statement in a subprogram is a call of another subprogram, this optimization implements the call as a jump to the callee, not as a call instruction. Bound-T makes an effort to detect such jumps and to model them as calls.

Parameter passing

By default SDCC passes the first parameter in registers (**DPL**, **DPH**, **B**, **A**) and the remaining parameters in statically allocated data memory. Bound-T can analyse scalar values that are passed by value in this way.

SDCC uses the same four registers to pass values returned by functions.

By default SDCC makes the calling subprogram save and restore all registers (**r0** .. **r7**), leaving the callee free to use and modify these registers. Compilation options or source-code pragmas can change this around so that the callee is responsible for saving and restoring registers. For an SDCC-compiled program Bound-T initially assumes that any call may alter any register; analysis of the callee may then reduce the set of altered registers. Thus Bound-T is not sensitive to the choice of caller-save or callee-save in the compilation phase.

Stacks

For reentrant subprograms SDCC can use the processor stack for parameters and local variables, in addition to return addresses. The option *--xstack* directs SDCC to use instead a software-managed stack in the paged part of the external data memory, where more space may be available. As explained above in section 7.1 Bound-T attempts to analyse computations that use data on the **SP** stack but does not do so for the optional external-memory stack.

Naked functions

SDCC supports the keyword *__naked* that declares a C function to be “naked” in the sense that the compiler provides no prologue or epilogue code. Instead, the function is itself responsible for saving and restoring registers, selecting the register bank, including a return instruction, and so on. Bound-T supports such code except that it cannot track changes in the choice of register bank.

Switch-case statements

Under investigation. Information to be provided in a later issue.

Library subprograms that violate the calling standard

Under investigation. Information to be provided in a later issue.

Symbolic debugging information

It seems that the SDDC compiler does not record the symbols for paged data variables in the CDB file. For such symbols the CDB file contains only the linker record that gives the absolute address of the variable, but not its type (size). For such symbols Bound-T by default assumes a type of “unsigned octet” and a location in the external memory. See the command-line option *-defxu8* in Table 5 on page 15.

Compiler options

Table 20 below lists those SDCC compiler options that influence the generated code and explains how Bound-T supports, or does not support, each option. Options not listed are supported.

If you have a pressing need to use an option that Bound-T does *not* support now, please contact Tidorum to discuss your needs.

Table 20: SDCC compiler options

Option	Supported?	Remarks
--all-callee-saves	Yes	Bound-T is not (currently) sensitive to the difference between caller-save or callee-save protocols.
--callee-saves	Yes	Bound-T is not (currently) sensitive to the difference between caller-save or callee-save protocols.
--debug	Yes, and recommended	Bound-T needs the debugging information to interpret assertions and parameters and to annotate the analysis results with source-code identifiers and locations.
--fdollars-in-identifiers	TBD	May cause trouble in CDB files where dollar signs are used as field delimiters.
--float-reent	TBD	TBD
--int-long-reent	TBD	TBD
--main-return	Yes	Without this option Bound-T will complain that the *main* function contains an eternal loop (the "lock up" loop [14]).
-m*P*	Only for *P = mcs51*	The only SDCC target processor that Bound-T currently supports is the MCS51 = 8051.
--model-small --model-medium	Yes	These *--model* options only affect the default memory space that the compiler chooses for variables and parameters.
--model-large	TBD	This *--model* option places variables in the external data memory by default, which is not a problem for Bound-T. However, [14] reports that the option also disables several optimizations, which might complicate the program code and hamper the analysis.
--nojtbound	**No**	Bound-T needs the boundary condition checks to find the bounds on the jump table.
--out-fmt-ihx	Yes	Bound-T can read Intel Hex format. Use a CDB file for the debugging information.
--out-fmt-s19	**No**	Bound-T for the 8051 cannot now read Motorola S19 format.
--stack-auto	Yes/**No**	This option makes the compiler place all parameters and local variables in a stack. Bound-T attempts to analyse computations using data in the SP stack.
--xstack	Yes/**No**	This option makes the compiler place stacked parameters and local variables in the external memory. At present Bound-T cannot analyse computations using such data.

7.5 Other compilers

If you want to use Bound-T to analyze programs compiled with some other cross-compilers for the 8051, please contact Tidorum to ask for advice. You can also check if your compiler can generate an executable in some form that Bound-T can read (for example Intel-Hex form, option *-form=ihex*), and if it can emit symbols for debugging in some form that can be translated to Bound-T symbol-definition files (option *-symbols filename*) as described in the Bound-T Reference Manual [2].

If the answer is yes to both questions, Bound-T should be able to analyse most code generated by your compiler, with possible problems (as always) for switch-case statements, function pointers, and library routines with special calling conventions.

8 WARNINGS AND ERRORS FOR THE 8051

8.1 Warning messages

The following Table 21 lists the Bound-T warning messages that are specific to the 8051 or that have a specific interpretation for this processor. The messages are listed in alphabetical order. Variable fields in the message are indicated by *italic* text and are ignored in the alphabetical ordering.

For other warning messages (not in Table 21) you can find explanations from other sources:

- The Bound-T Reference Manual [2] explains the generic warning messages, all of which may appear also when the 8051 is the target.
- The HRT-mode manual [5] explains the warnings that are specific to an HRT analysis.
- The Technical Note on AOMF [18] explains the warnings that are specific to AOMF program files.
- The Technical Note on Intel Hex [19] explains the warnings that are specific to Intel Hex program files.
- The Technical Note on the SDCC CDB format [22] explains the warnings that are specific to CDB symbol files from the SDCC compilation system.

Warnings specific to UBROF or (A)OMF2 files are not documented as UBROF and OMF2 are proprietary and closed formats. If such warnings appear please contact Tidorum.

Table 21: Warning messages

Warning Message		Meaning and Remedy
Assuming no misses in the CC2510 flash-cache	*Reasons*	Bound-T does not model the flash cache in the Texas Instruments CC2510 device. The execution-time model omits delays due to cache misses.
	Action	Note that the WCET bound for a CC2510 device is accurate only if the clock frequency is less than 13 MHz. For higher clock frequencies the WCET bound may be too small because delays from cache misses are omitted.
Assuming that Register Bank is NOT changed	*Reasons*	See the warning “PSW modified directly” below.
	Action	Ditto.
Call to address zero replaced by return at *A*	*Reasons*	The instruction at *A* is a call to address zero. Such a call results in a processor reset (reboot). Bound-T ends the analysis (of this execution path) at this call.
	Action	Take note that the given time and space bounds do not include the time for the reset and reboot.

Warning Message		Meaning and Remedy
CDB register not understood: *R*	*Reasons*	A symbol record in the CDB file [20] locates a variable in a register called *R*, but Bound-T does not undstand which 8051 register is meant by *R*. Bound-T ignores the symbol record. The CDB file may use a version of the CDB format [20] that Bound-T does not support [22] or it may be generated for another target processor. Fields 2 and 4 of the output line show the CDB file name and line number respectively.
	Action	Please report the problem to Tidorum.
Clearing PSW.P has no effect	*Reasons*	This instruction is **clr PSW.P**, that is, it clears the parity flag **P** in the **PSW**. However, the instruction has no effect because the next cycle makes **P** reflect the parity of **A**.
	Action	Consider why the program contains this useless instruction.
Code Banks (Seg_ID /= 0) not supported	*Reasons*	The program seems to use “code banking”. Bound-T does not support code banking.
	Action	Change the program to avoid code banking.
Complementing PSW.P has no effect	*Reasons*	This instruction is **cpl PSW.P**, that is, it inverts the parity flag **P** in the **PSW**. However, the instruction has no effect because the next cycle makes **P** reflect the parity of **A**.
	Action	Consider why the program contains this useless instruction.
Data not loaded into segment of type *T*	*Reasons*	The memory image of the targe program, as defined in the executable file, contains some data or code for a segment of type *T*, not supported in Bound-T.
	Action	Take note that the analysis does not use this data or code.
Move from a location to the same location	*Reasons*	This instruction is a **mov** where the source and destination are the same cell. Thus, the instruction has no effect, unless the cells are SFRs where reading and writing has some side effect on peripheral devices.
	Action	Check that the instruction is meaningful in this program.
PSW modified directly *followed by* Assuming that Register Bank is NOT changed	*Reasons*	An instruction stores a new value into the **PSW** (the whole octet). Bound-T assumes that this does not change the register bank selection.
	Action	Verify that Bound-T's assumption is correct. Otherwise take note that the analysis may fail to follow the flow of data within the general registers.
PSW Register Bank bits modified	*Reasons*	A bit (Boolean) instruction stores a new value into one of the register bank selection birs RS0 or RS1 in the PSW.
	Action	Take note that the analysis may fail to follow the flow of data within the general registers, because another register bank is selected.

Warning Message		Meaning and Remedy
Quit-flag ignored on return	*Reasons*	The partial evaluation of a special routine (such as a switch handler, see section 7.3, page 46) has resolved a return instruction into a normal return, not into a jump. In such a case, Bound-T continues the partial evaluation, rather than quitting it.
	Action	Check that the control-flow graph that results from the partial evaluation corresponds to the target program.
Setting PSW.P has no effect	*Reasons*	This instruction is **setb PSW.P**, that is, it sets the parity flag **P** in the **PSW**. However, the instruction has no effect because the next cycle makes **P** reflect the parity of **A**.
	Action	Consider why the program contains this useless instruction.
Stack initialized: sp = *V*	*Reasons*	This instruction assigns a statically known constant value *V* to the stack pointer **SP**. Bound-T assumes that this instruction initializes the stack area, and thus it assumes that the local stack height becomes zero at this point.
	Action	Note that any instructions using **SP** that occur earlier in this subprogram may be using a different stack area.
Store via pointer to bit-addressable octet at *A*	*Reasons*	This instruction stores a value in a memory location using a dynamically computed address (a pointer). Bound-T has resolved the address to *A* and noted that this is a bit-addressable memory octet. The current design of Bound-T means that the analysis ignores the effect of this instruciton on the one-bit cells within the octet at address *A*.
	Action	Note that the analysis of the values of the bits within the memory octet at *A* may be wrong. If these bits are important in the analysis of loop bounds or the feasibility of execution paths, the resulting execution-time bounds and stack-usage bounds may be wrong too.
The special routine *R* at *A* will be simulated	*Reasons*	The program executes a call or jump to a Keil library routine called *R*, with entry address *A*, and this routine has a special role which means that it must be analysed using simulation (partial evaluation), not as a normal subprogram. See section 7.3, page 46 for the Keil switch handler routines.
	Action	Note that the flow-graph of the (application) subprogram under analysis will be expanded to contain the result of the simulation of the special routine *R*.

8.2 Error messages

The following Table 22 lists the Bound-T error messages that are specific to the 8051 or that have a specific interpretation for this processor. The messages are listed in alphabetical order. Variable fields in the message are indicated by *italic* text and are ignored in the alphabetical order.

For other warning messages (not in this table) you can find explanations from other sources:

- The Bound-T Reference Manual [2] explains the generic error messages, all of which may appear also when the 8051 is the target.

- The HRT-mode manual [5] explains the error messages that are specific to an HRT analysis.
- The Technical Note on AOMF [18] explains the error messages that are specific to AOMF program files.
- The Technical Note on Intel Hex [19] explains the error messages that are specific to Intel Hex program files.
- The Technical Note on the SDCC CDB format [22] explains the error messages that are specific to CDB symbol files from the SDCC compilation system.

Error messages specific to UBROF or (A)OMF2 files are not documented as UBROF and OMF2 are proprietary and closed formats. If such errors appear please contact Tidorum.

Table 22: Error messages

Error Message		Meaning and Remedy
Address out of range for *M* memory: *A*	*Problem*	The analysis tried to use a memory location at address *A*, in memory of type *M* (Code, Internal Data, External Data, or Paged Data), but *A* is not in the range of addresses for this type of memory.
	Reasons	The debugging information in the program file wrongly claims that a variable in memory *M* has address *A*; or Bound-T has overestimated the possible addresses while analysing a dynamic memory reference (a memory reference with a computed address).
	Solution	Please inform Tidorum.
Cannot determine format of program file	*Problem*	In the absence of a *-form* option Bound-T has tried but failed to determine the format (type) of the target program file by examining the contents of the file itself.
	Reasons	Perhaps the program file has some other format, neither AOMF, AOMF2, Intel Hex, nor UBROF. Perhaps the program file is damaged.
	Solution	Make sure that your target program file has a supported format, and use a *-form* option to specify the format for Bound-T.
Cannot read file	*Problem*	The target program executable file is not readable.
	Reasons	The file may be read-protected (insufficient access rights) or the command-line may mistakenly name a directory or some special file that cannot be read.
	Solution	Correct the file access permissions or correct the file-name on the command line.
Code Banking is not supported	*Problem*	The program seems to be using code banking: an extension to the 8051 program memory architecture that allows more than 64 Ko of code. However, Bound-T does not currently support code banking.
	Reasons	The program is written and compiled with code banking enabled.

Error Message		Meaning and Remedy
	Solution	Try to make the program fit in 64 Ko without code banking. Perhaps divide the program (for analysis purposes) into smaller parts that do fit in 64 Ko.
File not found	*Problem*	The program file named on the command line was not found and could not be opened.
	Reasons	Perhaps the file name is mistyped; perhaps the file is in a directory that is not accessible.
	Solution	Correct the file name on the command line. Set directory access rights to allow access to the file.
Further CDB file ignored: *filename*	*Problem*	The command line has more than one *-cdb* option. This *filename* is mentioned in the second or later *-cdb* option and is therefore ignored; the file is not read.
	Reasons	Mistake on the command line.
	Solution	Correct the command line. If you have a persistent need to read several CDB files in the same run of Bound-T, please inform Tidorum.
Ignoring asserted “returns” values (must be single value 0 .. 2)	*Problem*	An assertion on the “returns” property specifies a range of values, or a single value outside the valid range 0 .. 2.
	Reasons	Error in the assertion.
	Solution	Correct the assertion. See section 3.7.
Invalid register number *R* for variable	*Problem*	The UBROF file claims that a program variable is stored in register number *R*, but *R* is not the number of an 8051 register, as Bound-T understands it.
	Reasons	Incompatible (too old or too new) version of UBROF, or an UBROF file compiled for another target processor.
	Solution	Ensure that the program is compiled for the 8051 and stored in a version of UBROF that Bound-T supports. If the problem persists, please inform Tidorum.
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 using some (device-specific) method; 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. The most common kind of error in the program-flow analysis is an over-estimation of the possible target addresses of a dynamic jump or call.
	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.

Error Message		Meaning and Remedy
Patching is not implemented for 8051	*Problem*	Bound-T cannot implement the command-line option *-patch* because program patching is not implemented for the 8051 target.
	Reasons	The *-patch* option is used on the command line.
	Solution	Remove the *-patch* option.
Paged-address base must be a multiple of 256	*Problem*	The base address given in the command-line option *-page=address* is not a multiple of 256 (octets), but it should be.
	Reasons	Error in command line.
	Solution	Correct the command line. The address given in *-page* must be a hexadecimal number with “00” as the last two digits.
Reading blank code memory at address *A*, using zero value	*Problem*	While analysing a **movc** instruction that reads data from code memory, Bound-T found that the content of the computed address *A* in code memory (which may be just one of the possible addresses) is not defined in the target program file. Had it been defined, Bound-T would have continued the analysis with the known code-memory value at this address, but now it will limp on with a zero value instead.
	Reasons	The target program file is incomplete; or the program itself stores something at this address at run-time using some (device-specific) method; or the program computes the **movc** address in a way that Bound-T cannot analyse, perhaps involving deliberate overflow.
	Solution	Disable arithmetic analysis of this subprogram. Assert bounds on the subprogram's loops.
Reading code memory at invalid address *A*, using zero value	*Problem*	While analysing a **movc** instruction that reads data from code memory, Bound-T found that the computed address *A* in code memory (which may be just one of the possible addresses) is out of range. Had it been in range, Bound-T would have continued the analysis with the known code-memory value at this address, but now it will limp on with a zero value instead.
	Reasons	The target program is probably computing the **movc** address in a way that Bound-T cannot analyse, perhaps involving deliberate overflow.
	Solution	Disable arithmetic analysis of this subprogram. Assert bounds on the subprogram's loops.
SP-relative offset too large: *D*	*Problem*	Bound-T is analysing an instruction that accesses memory indirectly using a dynamically computed address. Bound-T has determined that the address is computed by adding an offset *D* to the SP register. However, *D* is outside the valid range for 8-bit stack offsets.

Error Message		Meaning and Remedy
	Reasons	The program may be relying on overflow in its computation of the offset. For example, the offset may be computed as the sum of two positive numbers less than $2^8 = 256$ but giving a total of 256 or more. In the real processor, overflow will bring the total back below 256, but Bound-T's analysis does not consider overflow.
	Solution	Change the program to avoid such offset computations. Alternatively, disable arithmetic analysis of this subprogram, assert bounds on the subprogram's loops, and note that the stack usage computed for this subprogram may be incorrect.
Stack height at return is out of range: *H*	*Problem*	Bound-T is analysing a **ret** instruction as a possible dynamic jump. The first step in this analysis shows that the local stack height at this instruction is *H*. However, *H* is outside the valid range for 8-bit stack heights.
	Reasons	The program may be relying on overflow in its manipulation of **SP**. For example, it may be decreasing **SP** by adding a number greater than 127 to **SP**.
	Solution	Change the program to avoid such **SP** manipulations. Use **subb** to decrease **SP**. Alternatively, avoid using **ret** as a dynamic jump so that you can run Bound-T with the option *-returns=static*, the equivalent assertion on the "returns" property, or the equivalent instruction role assertion on this **ret**.
Stack offset out of range: *F*	*Problem*	While analysing a **pop** or **push** instruction (or some other reference to data on the stack) Bound-T finds that the computed offset *F*, relative to the stack pointer **SP**, is out of range.
	Reasons	The subprogram may be changing **SP** in some way that Bound-T cannot analyse, perhaps relying on overflow.
	Solution	Change the program to avoid such **SP** manipulations. Alternatively, disable arithmetic analysis of this subprogram, assert bounds on the subprogram's loops, and note that the stack usage computed for this subprogram may be incorrect.
The special routine *R* at *A* is not supported	*Problem*	The program executes a call or jump to a compiler library routine called *R*, with entry address *A*, and this routine has a special role which means that it cannot be analysed as a normal subprogram. In fact, Bound-T cannot analyse this routine at all.
	Reasons	At present this error message should not appear. However, this may change as Bound-T evolves.
	Solution	Please inform Tidorum of the problem.

Error Message		Meaning and Remedy
Truncated instruction	*Problem*	According to Bound-T's analysis, the program fetches a multi-octet instruction from a program memory address that contains only part of the instruction; that is, the target program file does not load code into all the octet addresses occupied by the instruction.
	Reasons	Same as for the error "No instruction loaded at this address", which see.
	Solution	Ditto.
Unacceptable paged address base: -paged=*B*	*Problem*	In this command-line *-page* option, the string *B* is not a valid (hexadecimal) base address for the paged addressing mode.
	Reasons	Error in command line.
	Solution	Correct the command line.
Unacceptable value for -movx:*C*	*Problem*	The value *C* specified as the number of cycles taken for a **movx** instruction in the nRF24E1 device is not an integer in the range 2 .. 9.
	Reasons	Error in command line option *-movx=C*.
	Solution	Correct the command line.
Unexpected end of file *or* Unexpected end of AOMF file *or* Unexpected end of Intel-Hex file *or* Unexpected end of UBROF file	*Problem*	The target program file (in AOMF, Intel-Hex, or UBROF form) is not complete; the file ended in the middle of reading it.
	Reasons	The target program file is damaged; or uses a file format or a format-variant that Bound-T cannot read; or Bound-T is trying to read the file with the wrong format, perhaps because of a mistake in a *-form* option.
	Solution	Obtain an undamaged program file, in a format that Bound-T can read, and use the correct *-form* option.
Unknown compiler: -compiler=*C*	*Problem*	In this command-line *-compiler* option, the string *C* is not the name of a cross-compiler that Bound-T knows about for the 8051.
	Reasons	Mistake in the command line.
	Solution	Correct the command line. See Table 4.
Unknown file format: -form=*F*	*Problem*	In this command-line *-form* option, the string *F* is not the name of a program-file format for the 8051 that Bound-T knows about.
	Reasons	Mistake in the command line.
	Solution	Correct the command line. See Table 6.
Unknown model for "returns": -returns=*R*	*Problem*	In this command-line *-returns* or *-return* option, the string *R* is not one of the values that Bound-T accepts: *dynamic* or *static*.
	Reasons	Mistake in the command line.
	Solution	Correct the command line.

Error Message		Meaning and Remedy
Unknown or invalid SP net change: -spch=*amount*	*Problem*	In this command-line *-spch* option, the string *amount* is not a numeric literal (perhaps with a leading sign), or the number is out of range (the **SP** stack is less than 256 octets).
	Reasons	Mistake in the command line.
	Solution	Correct the command line.
Unknown register bank: -reg_bank=*B*	*Problem*	In this command-line *-reg_bank* option, the string *B* is not the number of a register bank in the 8051, that is, not a number from 0 to 3.
	Reasons	Mistake in the command line.
	Solution	Correct the command line.
Variable address has no memory symbol: "*A*"	*Problem*	In an assertion that refers to a variable by its address *A* the address string does not start with a symbol for the memory space, followed by a colon.
	Reasons	Error in the assertion; wrong syntax for variable address.
	Solution	Correct the assertion. See section 3.4.
Variable address is too short: "*A*"	*Problem*	In an assertion that refers to a variable by its address *A* the address string is too short to be interpreted as the address of a variable in the 8051.
	Reasons	Error in the assertion; wrong syntax for variable address.
	Solution	Correct the assertion. See section 3.4.
Variable address not understood: "*A*"	*Problem*	In an assertion that refers to a variable by its address *A* the address string does not have the right form or specifies an address that is out of range for this memory space.
	Reasons	Error in the assertion; wrong syntax for variable address.
	Solution	Correct the assertion. See section 3.4.
Variable address starts with unknown memory symbol: "*A*"	*Problem*	In an assertion that refers to a variable by its address string *A* the first character, which should be a symbol for the memory space, is not one of the known memory symbols C, D, X, B.
	Reasons	Error in the assertion; wrong syntax for variable address.
	Solution	Correct the assertion. See section 3.4.

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