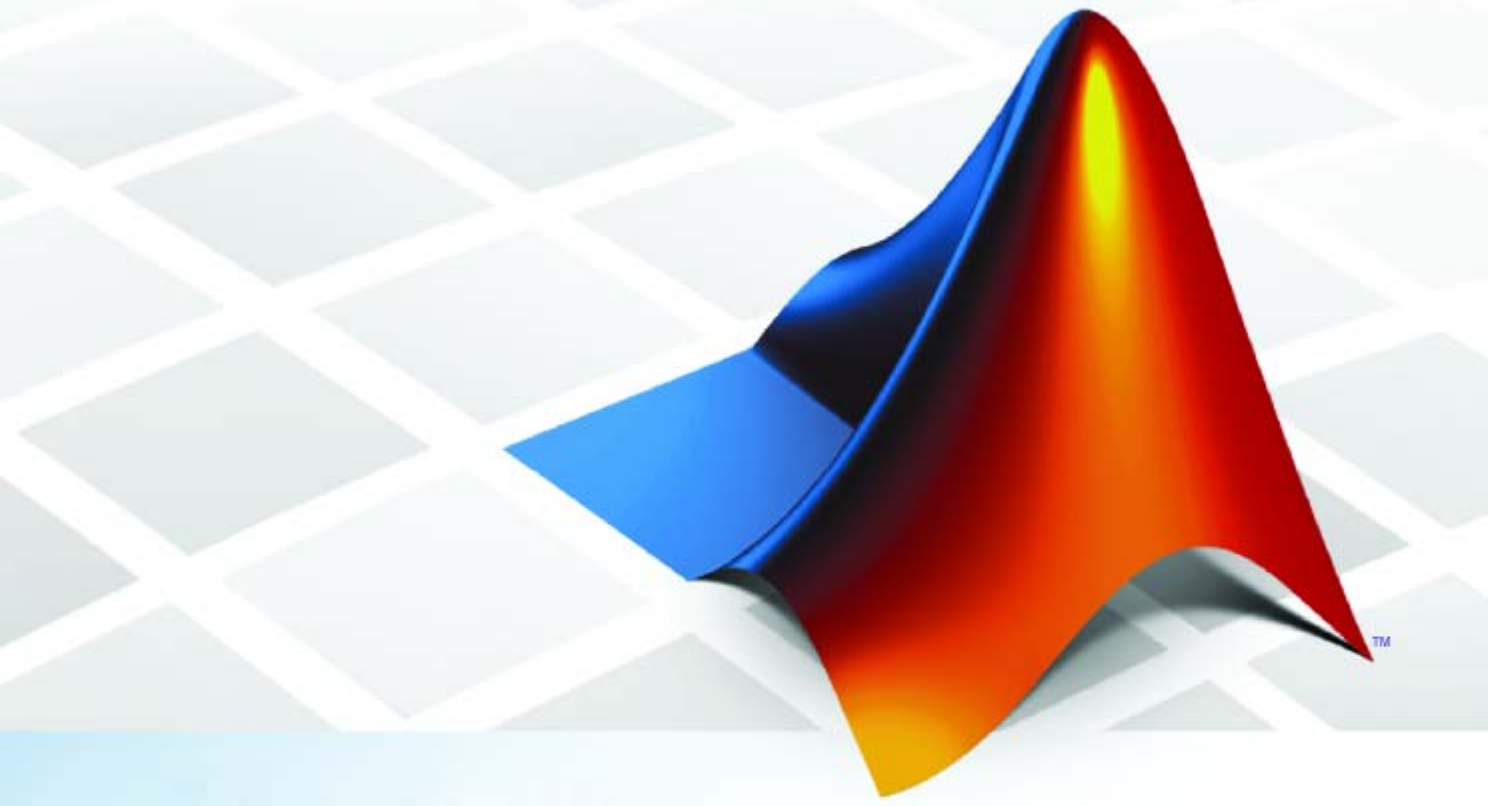
PolySpace® Products for C 7

Reference



The MathWorks™

Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

PolySpace® Products for C Reference

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

March 2009	Online Only	New for Version 7.0 (Release 2009a)
September 2009	Online Only	New for Version 7.1 (Release 2009b)

Contents

1 Option Descriptions

Check Descriptions

2

Approximations Used During Verification

3

4 Examples

1

Option Descriptions

General Options

In this section...

Overview

This section collates all options relating to the identification of the verification, including the destination directory for the results and sources.

-prog Session identifier

This option specifies the application name, using only the characters which are valid for Unix file names. This information is labelled in the GUI as the *Session Identifier*.

Default:

Shell Script: polyspace

GUI: New_Project

Example shell script entry:

```
polyspace-c -prog myApp ...
```

-date Date

This option specifies a date stamp for the verification in dd/mm/yyyy format. This information is labelled in the GUI as the *Date*. The GUI also allows alternative default date formats, via the Edit/Preferences window.

Default:

Day of launching the verification

Example shell script entry:

```
polyspace-c -date "02/01/2002"...
```

-author Author

This option is used to specify the name of the author of the verification.

Default:

the name of the author is the result of the *whoami* command

Example shell script entry:

```
polyspace-c -author "John Tester"
```

-verif-version Version

Specifies the version identifier of the verification. This option can be used to identify different verifications. This information is identified in the GUI as the *Version*.

Default:

1.0.

Example shell script entry:

```
polyspace-c -verif-version 1.3 ...
```

-keep-all-files

When this option is set, all intermediate results and associated working files are retained. Consequently, it is possible to restart PolySpace® verification from the end of any complete pass (provided the source code remains entirely unchanged). If this option is not used, you must restart the verification from scratch.

By default, intermediate results and associated working files are erased when they are no longer needed by the software.

This option is applicable only to client verifications. Intermediate results are always removed before results are downloaded from the PolySpace server.

Note To cleanup intermediate files at a later time, you can select **Tools > Clean Results** in the Launcher.

This options deletes the preliminary results files from the results directory.

Default:

Disabled.

Example shell script entry:

```
polyspace-c -keep-all-files
```

-continue-with-red-error (Deprecated)

Note This option is deprecated in R2009a and later releases, and no longer exists in the user interface. Verification now continues to the next integration pass even if a red errors is encountered.

This option allows PolySpace verification to continue even if one of these red errors is encountered. In most cases, this will mean that the dynamic behavior of the code beyond the point where red errors are identified will be undefined, unless the red code is actually inaccessible.

-continue-with-existing-host

When this option is set, the verification will continue even if the system is under specified or its configuration is not as preferred by PolySpace software. Verified system parameters include the amount of RAM, the amount of swap space, and the ratio of RAM to swap.

Default:

verification stops when the host configuration is incorrect or the system is under specified.

Example Shell Script Entry:

```
polyspace-c -continue-with-existing-host ...
```

-allow-unsupported-linux

This option specifies that PolySpace verification will be launched on an unsupported OS Linux® distribution.

PolySpace software supports the Linux distributions listed in "Hardware and Software Requirements" in the *PolySpace Installation Guide*.

For all other Linux distributions, you may be able to verify code using the `-allow-unsupported-linux` option, but a warning will be displayed in the log file informing you of possible incorrect behaviors:

```
************************************************** ***
***                    ***
***         WARNING          ***
***                    ***
***   You are running PolySpace on an   ***
***   unsupported Linux distribution. It may lead  ***
***   to incorrect behaviour of the product. Please ***
***   note that no support will be available for  ***
***   this operating system.          ***
***                    ***
************************************************ ** ***
```

Default:

Disabled

Example Shell Script Entry:

```
polyspace-c  allow-unsupported-linux ...
```

Report Generation

When this option is selected, PolySpace software creates a report for the verification, using the following options:

- "-report-template Report_Template_Name" on page 1-6
- "-report-output-format Output_Format" on page 1-7
- "-report-output-name Name" on page 1-7

-report-template Report_Template_Name

Generates a report for the verification, using the specified report template name. The report is generated at the end of the verification process, before any `post-analysis-command` is executed.

Default:

```
C:\PolySpace\PolySpace_Common\ReportGenerator\templates\Developer.rpt
```

Example Shell Script Entry:

```
polyspace-c -report-template c:/polyspace/my_template
```

-report-output-format Output_Format

Specifies the output format for the report specified by the `report-template` option. The argument is not case sensitive.

Valid options are:

- HTML
- PDF
- RTF
- WORD
- XML

Note WORD format is not available on UNIX platforms, RTF format is used instead.

Default:

If you do not specify an output format, RTF is used by default.

Example Shell Script Entry:

```
polyspace-c -report-template my_template  report-output-format
pdf
```

-report-output-name Name

Specifies the name of the report file that is generated for the verification.

Default:

If you do not specify a name, the following name is used by default:

-Prog_TemplateName.Format

where *Prog* is the argument of prog option, *TemplateName* is the name of the report template specified by the -report-template option, and *Format* is the file extension for the format specified by the report-output-format option.

Example Shell Script Entry:

```
polyspace-c -report-template my_template  report-output-name
Airbag_V3.rtf
```

-results-dir Results Directory

This option specifies the directory in which PolySpace software will write the results of the verification. Note that although relative directories may be specified, particular care should be taken with their use especially where the tool is to be launched remotely over a network, and/or where a project configuration file is to be copied using the "Save as" option.

Default:
Shell Script: The directory in which tool is launched.
From Graphical User Interface: C:\PolySpace_Results

Example Shell Script Entry:

```
polyspace-c -results-dir RESULTS ...
export RESULTS=results_`date +%d%B_%HH%M_%A`
polyspace-c -results-dir `pwd`/$RESULTS ...
```

-sources "files" or -sources-list-file file_name

Specifies a list of source files to be verified.

The list of source files must be double-quoted and separated by commas.

- -sources "*file1*[*file2*[...]]" (Linux and Solaris™)
- -sources "*file1*[,*file2*[, ...]]" (Windows®, Linux and Solaris)
- -sources-list-file *file_name* (not a graphical option)

Note UNIX® standard wild cards are available to specify a number of files.

The source files are compiled in the order in which they are specified.

Note If you do not specify any files, the software verifies all files in the source directory in alphabetical order.

Note The specified files must have valid extensions:
*.(c|C|cc|cpp|CPP|cxx|CXX)

Defaults:

```
sources/*.(c|C|cc|cpp|CPP|cxx|CXX)
```

Example Shell Script Entry under linux or solaris (*files are separated with a white space)*:

```
polyspace-c -sources "my_directory/*.cpp" ...
polyspace-c -sources "my_directory/file1.cc other_dir/file2.cpp"
...
```

Example Shell Script Entry under windows (*files are separated with a comma):*

```
polyspace-c -sources "my_directory/file1.cpp,other_dir/file2.cc"
...
```

Using `-sources-list-file`, each file *name* need to be given with an absolute path. Moreover, the syntax of the file is the following:

- One file by line.
- Each file name is given with its absolute path.

Note This option is only available in batch mode

Example Shell Script Entry for -sources-list-file:

```
polyspace-c -sources-list-file "C:\Analysis\files.txt"
polyspace-c -sources-list-file "/home/poly/files.txt"
```

-I directory

This option is used to specify the name of a directory to be included when compiling C sources. Only one directory may be specified for each –I, but the option can be used multiple times.

Default:

- When no directory is specified using this option, the ./sources directory (if it exists) is automatically included
- If several include-dir are mentioned, the ./sources directory (if it exists), is implicitly added at the end of the "-I" list

Example Shell Script Entry-1:

```
polyspace-c -I /com1/inc -I /com1/sys/inc
```

is equivalent to

```
polyspace-c -I /com1/inc -I /com1/sys/inc -I ./sources
```

Example Shell Script Entry-2:

```
polyspace-c
```

is equivalent to

```
polyspace-c -I ./sources
```

Target/Compiler Options

In this section...

Overview

This section allows details of the target processor and operating system to be specified. Header files should not be entered here; instead, include directories should be added using the relevant field under the Compile flag options.

-target TargetProcessorType

This option specifies the target processor type, and in doing so informs the verification of the size of fundamental data types and of the endianess of the target machine.

Possible values are: sparc, m68k, powerpc, i386, c-167, tms320c3x, sharc21x61, necv850, mcpu, or generic target.

mcpu is a reconfigurable Micro Controller/Processor Unit target. One or more generic target can also be specified and saved. Also code which is to be run on an unlisted processor type can be analyzed using one of the other processor types listed, if the data properties which are relevant to PolySpace verification are common. For more information, see "Setting Up Project for Generic Target Processors" in the *PolySpace Products for C User's Guide*.

Instructions on the specification of a generic target and on the modification of the mcpu target are available in “GENERIC ADVANCED TARGET OPTIONS” on page 1-12.

Default:

sparc

Example shell script entry:

```
polyspace-c -target m68k ...
```

GENERIC ADVANCED TARGET OPTIONS

The previous *Generic target options* dialog box is only available when a *mcpu* target is selected. (*Enter the target name* in PolySpace Launcher)

Allows the specification of a generic *"Micro Controller/Processor Unit"* or *mcpu* target name. Initially, it is necessary to use the GUI to specify the name of a new *mcpu*target – say, “MyTarget”.

That new target is added to the -target options list. The new target’s default characteristics are as follows, using the *type [size, alignment]* format.

- *char [8, 8, char [16,16]]*
- *short [8,8], short [16, 16]*
- *int [16, 16]*
- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*
- *char is signed*
- *little-endian*

When using the command line, *MyTarget* is specified with all the options for modification:

```
polyspace-c -target MyTarget
```

For example, a specific target uses 8 bit alignment (see also -align), for which the command line would read:

```
polyspace-c -target mcpu -align 8
```

-little-endian

This option is only available when a *-mcpu* generic target has been chosen.

The endianness defines the byte order within a word (and the word order within a long integer). Little-endian architectures are Less Significant byte First (LSF), for example: i386.

For a little endian target, the less significant byte of a short integer (for example 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte.

Example shell script entry:

```
polyspace-c -target mcpu -little-endian
```

-big-endian

This option is only available when a *-mcpu* generic target has been chosen.

The endianness defines the byte order within a word (and the word order within a long integer). Big-endian architectures are Most Significant byte First (MSF), for example: SPARC, m68k.

For a big endian target, the most significant byte of a short integer (for example 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte.

Example shell script entry:

```
polyspace-c -target mcpu -big-endian
```

-default-sign-of-char [signed|unsigned]

This option is available for all targets. It allows a char to be defined as "signed", "unsigned", or left to assume the mcpu target's default behavior

- **default mode** – The sign of char is left to assume the target's default behavior. By default all targets are considered as signed except for hc08 and powerpc targets.
- **signed** – Disregards the target's default char definition, and specifies that a "signed char" should be used.
- **unsigned** – Disregards the target's default char definition, and specifies that a "unsigned char" should be used.

Example Shell Script Entry

```
polyspace-c -default-sign-of-char unsigned -target mcpu ...
```

-char-is-16bits

This option is only available when a *-mcpu* generic target has been chosen.

The default configuration of a generic target defines a char as 16 bits. This option changes it to 16 bits, regardless of sign.

the minimum alignment of objects is also set to 16 bits and so, incompatible with the options -short-is-8bits and -align 8.

Setting the char type to 16 bits has consequences on the following:

- computation of size of for objects
- detection of underflow and overflow on chars

Without the option char for *mcpu* are 8 bits

Example shell script entry:

```
polyspace-c -target mcpu -char-is-16bits
```

-short-is-8bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a short as 16 bits. This option changes it to 8 bits, regardless of sign.

It sets a short type as 8-bit without specific alignment. That has consequences for the following:

- computation of size of objects referencing short type
- detection of short underflow/overflow

Example shell script entry

```
polyspace-c -target mcpu -short-is-8bits
```

-int-is-32bits

This option is available with a *mcpu* generic target, hc08, hc12 and mpc5xx target has been chosen.

The default configuration of a generic target defines an int as 16 bits. This option changes it to 32 bits, regardless of sign. Its alignment, when an int is used as struct member or array component, is also set to 32 bits. See also -align option.

Example shell script entry

```
polyspace-c -target mcpu -int-is-32bits
```

-long-long-is-64bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a long long as 32 bits. This option changes it to 64 bits, regardless of sign. When a long long is used as struct member or array component, its alignment is also set to 64 bits. See also -align option.

Example shell script entry

```
polyspace-c -target mcpu -long-long-is-64bits
```

-double-is-64bits

The default configuration of a generic target defines a double as 32 bits. This option, changes both double and *long double* to 64 bits. When a double or long double is used as a struct member or array component, its alignment is set to 4 bytes.

See also -align option.

Defining the double type as a 64 bit double precision float impacts the following:

- Computation of sizeofobjects referencing double type
- Detection of floating point underflow/overflow

This option is available for the following targets:

- *mcpu* generic target
- sharc21x61
- hc08
- hc12
- mpc5xx

Example

```
int main(void)
{
 struct S {char x; double f;};
 double x;
 unsigned s1, s2;
 s1 = sizeof (double);
 s2 = sizeof(struct S);
 x = 3.402823466E+38; /* IEEE 32 bits float point maximum value */
 x = x * 2;
 return 0;
}
```

Using the default configuration of sharc21x62, PolySpace verification assumes that a value of 1 is assigned to s1, 2 is assigned to s2, and there is a consequential float overflow in the multiplication x * 2. Using the –double-is-64bits option, a value of 2 is assigned to s1, and no overflow occurs in the multiplication (because the result is in the range of the 64-bit floating point type)

Example shell script entry

```
polyspace-c -target mcpu -double-is-64bits
```

-pointer-is-32bits

This option is only available when a *mcpu* generic target has been chosen.

The default configuration of a generic target defines a pointer as 16 bits. This option changes it to 32 bits. When a pointer is used as struct member or array component, its alignment is also set also to 32 bits (see -align option).

Example shell script entry

```
polyspace-c -target mcpu -pointer-is-32bits
```

-align [8|16|32]

This option is available with a *mcpu* generic target and some other specific targets (with hc08, hc12 or mpc5xx available values are 16 and 32). It is used to set the largest alignment of all data objects to 4/2/1 byte(s), meaning a 32, 16 or 8 bit boundary respectively.

-align 32 (Default). The default alignment of a generic target is 32 bits. This means that when objects with a size of more than 4 bytes are used as struct members or array components, they are aligned at 4 byte boundaries.

Example shell script entry with a 32 bits default alignment

```
polyspace-c -target mcpu
```

-align 16. If the `-align 16` option is used, when objects with a size of more than 2 bytes are used as struct members or array components, they are aligned at 2 bytes boundaries.

Example shell script entry with a 16 bits specific alignment:

```
polyspace-c -target mcpu -align 16
```

-align 8. If the `-align 8` option is used, when objects with a size of more than 1 byte are used as struct members or array components, are aligned at 1 byte boundaries. Consequently the storage assigned to the arrays and structures is strictly determined by the size of the individual data objects without member and end padding.

Example shell script entry with a 8 bits specific alignment:

```
polyspace-c -target mcpu -align 8
```

-logical-signed-right-shift

In the Graphical User Interface, the user can choose between arithmetical and logical computation.

- **- Arithmetic:** the sign bit remains:

```
(-4) >> 1 = -2
(-7) >> 1 = -4
 7 >> 1 = 3
```

- **- Logical:** 0 replaces the sign bit

```
(-4) >> 1 = (-4U) >> 1 = 2147483646
(-7) >> 1 = (-7U) >> 1 = 2147483644
 7 >> 1 = 3
```

Example shell script entry

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation will be performed.

```
polyspace-c -logical-signed-right-shift
```

-OS-target OperatingSystemTargetForPolySpaceStubs

This option specifies the operating system target for PolySpace stubs.

Possible values are 'Solaris', 'Linux', 'VxWorks', 'Visual' and 'no-predefined-OS'.

This information allows the appropriate system definitions to be used during preprocessing in order to analyze the included files properly. *-OS-target* no-predefined-OS may be used in conjunction with -include or/and -D to give all of the system preprocessor flags to be used at execution time. Details of these may be found by executing the compiler for the project in verbose mode. They are also listed in this document - search for keyword "OS-target option"

Default:

Solaris

Note Only the Linux include files are provided with PolySpace software (see the include folder in the installation directory). Projects developed for use with other operating systems may be analyzed by using the corresponding include files for that OS. For instance, in order to analyze a VxWorks® project it is necessary to use the option -I <<path_to_the_VxWorks_include_folder>>

Example shell script entry:

```
polyspace-c -OS-target linux
polyspace-c  -OS-target no-predefined-OS -D GCC_MAJOR=2     /
      -include /complete_path/inc/gn.h  ...
```

-D compiler-flag

This option is used to define macro compiler flags to be used during compilation phase.

Only one flag can be used with each –D as for compilers, but the option can be used several times as shown in the example below.

Default:

Some defines are applied by default, depending on your -OS-target option.

Example Shell Script Entry:

```
polyspace-c -D HAVE_MYLIB -D USE_COM1 ...
```

-U compiler-flag

This option is used to undefine a macro compiler flags

As for compilers, only one flag can be used with each –U, but the option can be used several times as shown in the example below.

Default:

Some undefines may be set by default, depending on your -OS-target option.

Example Shell Script Entry:

```
polyspace-c -U HAVE_MYLIB -U USE_COM1 ...
```

-include file_name

This option is used to specify files to be included by each C file involved in the verification.

Default:

No file is universally included by default, but directives such as "#include <*include_file.h*>" are acted upon.

Example Shell Script Entry:

```
polyspace-c -include `pwd`/sources/a_file.h -include
/inc/inc_file.h ...

polyspace-c -include /the_complete_path/my_defines.h ...
```

-post-preprocessing-command <file_name> or "command"

When this option is used, the specified script file or command is run just after the preprocessing phase on each source file. The script executes on each preprocessed c file. The command should be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.

Note You can find each preprocessed file in the results directory in the zipped file ci.zip located in <results/ALL/SRC/MACROS. The extension of the preprocessed file is .ci.

It is important to preserve the number of lines in the preprocessed .ci file. Adding a line or removing one could result in some unpredictable behavior on the location of checks and MACROS in the PolySpace viewer.

Default:

No command.

Example Shell Script Entry – file name:

To replace the keyword “Volatile” by “Import”, you can type the following command on a Linux workstation:

```
polyspace-c -post-preprocessing-command `pwd`/replace_keywords
```

where `replace_keywords` is the following script:

```
#!/usr/bin/perl
my $TOOLS_VERSION = "V1_4_1";
binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{
  # Change Volatile to Import
  $line =~ s/Volatile/Import/;
```

```
    print $line;
  }
```

Note If you are running PolySpace software version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin™ shell scripts. Since Cygwin is no longer included with PolySpace software, all files must be executable by Windows. To support scripting, the PolySpace installation now includes Perl. You can access Perl in

```
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
```

To run the Perl script provided in the previous example on a Windows workstation, you must use the option `-post-preprocessing-command` with the absolute path to the Perl script, for example:

```
%POLYSPACE_C%\Verifier\bin\polyspace-c.exe
-post-preprocessing-command
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\replace_keywords
```

-post-analysis-command <file_name> or "command"

When this option is used, the specified script file or command is executed once the verification has completed.

The script or command is executed in the results directory of the verification.

Execution occurs after the last part of the verification. The last part of is determined by the –to option.

Note Depending of the architecture used (notably when using remote launcher), the script can be executed on the client side or the server side.

Default:

No command.

Example Shell Script Entry – file name:

This example shows how to send an email to tip the client side off that his verification has been ended. So the command looks like:

```
polyspace-c -post-analysis-command `pwd`/end_emails
```

where `end_emails` is an appropriate Perl script.

Note If you are running PolySpace software version 5.1 (r2008a) or later on a Windows system, you cannot use Cygwin shell scripts. Since Cygwin is no longer included with PolySpace software, all files must be executable by Windows. To support scripting, the PolySpace installation now includes Perl. You can access Perl in

```
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
```

To run the Perl script provided in the previous example on a Windows workstation, you must use this option with the absolute path to the Perl script, for example:

```
%POLYSPACE_C%\Verifier\bin\polyspace-c.exe -post-analysis-command
%POLYSPACE_C%\Verifier\tools\perl\win32\bin\perl.exe
<absolute_path>\end_emails
```

Compliance with Standards Options

In this section...

-dos

This option must be used when the contents of the **include** or **source** directory comes from a DOS or Windows file system. It deals with upper/lower case sensitivity and control characters issues.

Concerned files are:

- header files: all include dir specified (-I option)
- source files: all sources files selected for the verification (-sources option)

```
#include "..\mY_TEst.h"^M

#include "..\mY_other_FILE.H"^M
```

into

```
#include "../my_test.h"

#include "../my_other_file.h"
```

Default:

disabled by default

Example Shell Script Entry:

polyspace-c -I /usr/include -dos -I ./my_copied_include_dir -D test=1

Embedded Assembler

-discard-asm

This option instructs the PolySpace verification to discard assembler code. If this option is used, the assembler code should be modelled in c.

Default:

Embedded assembler is treated as an error.

Example Shell Script Entry:

```
polyspace-c -discard-asm ...
```

Pragmas asm

```
-asm-begin "mark1[mark2[...]]  "
```

and

```
-asm-end "mark1[mark2[...]]"
```

These options are used to allow compiler specific asm functions to be excluded from the verification, with the offending code block delimited by two #pragma directives.

Consider the following example.

```
  #pragma asm_begin_1
```

```
int foo_1(void) { /* asm code to be ignored by PolySpace */ }
#pragma asm_end_1
#pragma asm_begin_2
void foo_2(void) { /* asm code to be ignored by PolySpace */ }
#pragma asm_end_2
```

Where "asm_begin_1" and "asm_begin_2" marks the beginning of asm sections which will be discarded and "asm_end_1", respectively "asm_end_2" mark the end of those sections.

Also refer to the -discard-asm option with regards to the following code:

```
asm int foo_1(void) { /* asm code to be ignored by PolySpace */ }

asm void foo_2(void) { /* asmcode to be ignored by PolySpace */ }
```

Note The asm-begin and asm-end options must be used together.

Example Shell Script Entry:

```
polyspace-c -discard-asm -asm-begin "asm_begin_1,asm_begin_2"
-asm-end "asm_end_1,asm_end_2" ...
```

Strictness during verification launching

-strict

This option selects the Strict mode of PolySpace verification. It is equivalent to using the -Wall and -no-automatic-stubbingoptions simultaneously.

This option is not compatible with -asm-begin and -asm-end options.

-wall

When this option is used, the C compliance phase will print all warnings. For example, with this option, a warning will raise in the log file during compilation phase when trying to write into a const variable: “warning: assignment of read-only member <var>”

Default:

By default, only warnings about compliance across different files are printed.

Example Shell Script Entry:

```
polyspace-c -Wall ...
```

Permissiveness during verification launching

-permissive

This option selects the PolySpace permissive mode, which is equivalent to the simultaneous use of -allow-non-int-bitfield, -allow-undef-variables, -ignore-constant-overflows, -discard-asm, -permissive-stubber, -continue-with-red-error, and -permissive-link.

-permissive-link

When this option is used, PolySpace verification accepts integral type conflicts between declarations and definitions on arguments or/and returning functions.

It has an effect only

- when the size of a conflicting integral type is not greater than int, or
- conflicts occur between a pointer type and an integral type of same size.

Default:

By default, PolySpace verification does not accept any conflicts between declarations and definitions.

-allow-non-int-bitfield

This option allows the user to define types of bitfields other than those specified by ANSI® C. The standard accepts bitfields of signed and unsigned int types only.

Default:

Bitfields must be signed or unsigned int.

Example Shell Script Entry :

```
polyspace-c -allow-non-int-bitfield ...
```

-allow-undef-variables

When this option is used, PolySpace verification will continue in case of linkage errors due to undefined global variables. For instance when this option is used, PolySpace verification will tolerate a variable always being declared as extern

Default:

Undefined variables causes PolySpace verification to stop.

Example Shell Script Entry:

```
polyspace-c -allow-undef-variables ...
```

-ignore-constant-overflows

This option specifies that the verification should be permissive with regards to overflowing computations on constants. Note that it deviates from the ANSI C standard.

For example,

```
char x = 0xff;
```

causes an overflow according to the standard, but if it is analyzed using this option it becomes effectively the same as

```
char x = -1;
```

With this second example, a red overflow will result regardless of the use of the option.

char x = (rnd?0xFF:0xFE);

Default:

`char x = 0xff;` causes an overflow

Example Shell Script Entry:

```
polyspace-c -ignore-constant-overflows ...
```

-allow-unnamed-fields

When this option is used, PolySpace verification will continue in case of compilation errors due to unnamed fields in structures. For instance when this option is used, PolySpace verification will tolerate a structure where fields are unnamed since there are no duplicate names. With the option, the following source code is tolerate:

```
struct {
```

```
 union { int x; int y;}
 union {int z; int w;}
} s;
s.x = 2; s.z = 2;
```

Default:

Unnamed fields cause PolySpace to stop.

Example Shell Script Entry:

```
polyspace-c -allow-unnamed-fields ...
```

-allow-negative-operand-in-shift

This option permits a shift operation on a negative number.

According to the ANSI C standard, such a shift operation on a negative number is illegal – for example,

-2 << 2

With this option in use, PolySpace verification considers the operation to be valid. In the example, the result would be

-2 << 2 = -8

Default:

A shift operation on a negative number causes a red error.

Example Shell Script Entry:

```
polyspace-c -allow-negative-operand-in-shift ...
```

MISRA-C 2004 Rules

- " -misra2 [all-rules | file_name]" on page 1-31

- "-includes-to-ignore "dir_or_file_path1[,dir_or_file_path2[,...]]"" on page 1-31

-misra2 [all-rules | file_name]

This option permits to check set of coding rules in conformity to MISRA-C:2004. All MISRA checks are included in the log file of the verification.

- Keyword *all-rules:* It checks all available MISRA C® rules. It implies the use of the default configuration: any violation of MISRA C rules is considered as a warning.
- Option *filename:* it is the name of an absolute ASCII file containing a list of MISRA® rules to check.

Format of the file:

```
<rule number> off|error|warning
# is considered as comments.
Example:
# MISRA configuration file for project C89
10.5 off # disable misra rule number 10.5
17.2 error # violation misra rule 17.2 as an error
17.3 warning # non-respect to misra rule 17.3 is a only a warning
```

Default:

disable

Example shell script entry:

```
polyspace-c -misra2 all-rules ...

polyspace-c -misra2 misra.txt
```

-includes-to-ignore "dir_or_file_path1[,dir_or_file_path2[,...]]"

This option prevents MISRA rules checking in a given list of files or directories (all files and subdirectories under selected directory). This option is useful

when non-MISRA C conforming include headers are used. A warning is displayed if one of the parameter does not exist.

This option is authorized only when -misra2 is used.

Example shell script entry :

```
   polyspace-c -misra2 misra.txt  includes-to-ignore
"c:\usr\include"
```

-dialect [iar|keil]

When this option is used, PolySpace verification will take into account some non Target Support Package™ syntax and semantic associated to a chosen dialect between IAR and Keil. It refers to the well known compilers of the company IAR (www.iar.com) and Keil (www.keil.com).

Using this option, PolySpace verification will tolerate some new structure types as keyword of the language such sfr, sbit, bit etc. These structures and associated semantics are part of the compiler that have integrated it with the ANSI C language as an extension.

Example of source code with keil dialect:

```
unsigned char bdata Status[4];
sfr AU = 0xF0;
sbit OCmd = Status[0]^2;
s^2 = 1; s^6 = 0;
```

Example with iar dialect:

```
unsigned char bdata Status[4];
sfr OCmd @ 0x4FFE;
OCmd.2 = 1; s.6 = 0;
```

Example Shell Script Entry:

```
polyspace-c  dialect keil
```

-sfr-types

Associated to the option -dialect, if the code uses specific sfr type keyword, it is **mandatory**to declare using –**sfr-types** option. It gives the name of the sfr type and its size in bits. The syntax is:

```
-sfr-types <sfr_name>=<size_in_bits>,
```

where <sfr_name> could be any name, but most of the time we encounter sfr, sfr16 and sfr32 . <size in bits> could be one of the values 8, 16 and 32.

Default:

No dialect used.

Example Shell Script Entry:

```
polyspace-c  dialect iar  sfr-types sfr=8,sfr32=32,sfrb=16
```

PolySpace Inner Settings Options

In this section...

-unit-by-unit

This option creates a separate verification job for each source file in the project.

Each file is compiled, sent to the PolySpace Server, and verified individually. Verification results can be viewed for the entire project, or for individual units.

Note Unit by unit verification is available only for server verifications. It is not compatible with multitasking options such as `-entry-points`.

Default:

Not selected

Example Shell Script Entry:

```
polyspace-c -unit-by-unit
```

-unit-by-unit-common-source *filename*

Specifies a list of files to include with each unit verification. These files are compiled once, and then linked to each unit before verification. Functions not included in this list are stubbed.

Default:

None

Example Shell Script Entry:

```
polyspace-c -unit-by-unit-common-source
c:/polyspace/function.c
```

MAIN GENERATOR OPTIONS (-main-generator) for PolySpace Software

This same option can be used for both PolySpace® Client™ for C/C++ and PolySpace® Server™ for C/C++, but the default behavior differs between the two:

- **Using PolySpace Server** the user has the choice as to whether to activate the option.
- **Using PolySpace Client** the option is activated by default.

This section describes:

PolySpace Client for C/C++ default behavior

There is no need to ascertain whether the code for verification contains a "main" or not. That is automatically checked by the PolySpace Client for C/C++ product:

- If a main exists in the set of file(s), then the verification proceeds with that main.
- Otherwise, the tool generates a main with default options: -main-generator-writes-variables public and -main-generator-calls unused.

PolySpace Server for C/C++ default behavior

By default, if no main is found in a PolySpace Server for C/C++ verification, then it will stop. This behavior can help isolate files missing from the verification.

It is also possible to allow the PolySpace Server for C/C++ product to ascertain whether or not a main is available.

- If an available main is found, the verification proceeds as usual.
- Otherwise, the tool generates a main with the assumption of verifying a library. The options used are -main-generator-writes-variables none and -main-generator-calls none.

-main-generator (detailed options)

This option initiates the default behavior for PolySpace Launcher. The generated main has three distinct default behaviors.

- It first initializes any variables identified by the option -main-generator-writes-variables. The default setting for this option is -main-generator-writes-variables public.
- It then calls a function which could be considered an initialization function with the option -function-called-before-main.
- It then calls any functions identified by the option -main-generator-calls. The default setting for this option is -main-generator-calls unused.

What follows are separate descriptions of the above options that include all of the possible settings.

-main-generator-writes-variables [none | public | all | custom=v1,v2,.]

This option is used with the -main-generator option to dictate how the generated main will initialize global variables.

Settings available:

- `-none` — no global variable will be written by the main.
- `-public` — every variable except static and const variables are assigned a "random" value, representing the full range of possible values
- `-all` — every variable except const variables are assigned a "random" value, representing the full range of possible values
- `-custom` — only variables present in the list are assigned a "random" value, representing the full range of possible values

Example

```
polyspace-c -main-generator -main-generator-writes-variables none
polyspace-c -main-generator -main-generator-writes-variables
custom=variable_a,variable_b
```

-function-called-before-main function_name

It is possible to specify an initialization function that will be called on startup after the initialization of the global variables and before the main loop when using the -main-generator option.

The skeleton of the generated main looks like:

1. Initialization of global variables
2. Call the specified function fname
3. main loop with a call to all the specified functions depending on option -main-generator-calls

Example shell script entry:

```
polyspace-c -main-generator  function-called-before-main
MyInitFunction
```

-main-generator-calls [none | unused | all | custom=f1,f2,...]

The generated main will call functions according to this option. It is used with the -main-generator option, to specify the functions to be called.

Possible values:

- `none` — no function is called. This can be used with a multitasking application without a main.
- `unused` (default) — every function is called by the generated main unless it is called elsewhere by the code undergoing verification.
- `all` — every function is called by the generated main except inlined.
- `custom` — only functions present in the list are called from the main. Inlined functions can be specified in the list.

An inline (static or extern) function is not called by the generated main program with values all or unused. An inline function can only be called with custom value: `-main-generator-calls custom=`*`my_inlined_func`*.

Note When using the `unused` option, the generated main may call functions that are also called by a function pointer, meaning these functions may be called twice.

Example:

```
polyspace-c -main-generator -main-generator-calls public

polyspace-c -main-generator -main-generator-calls
custom=function_1,function_2
```

Stubbing

-data-range-specifications file_name

This option permits the setting of specific data ranges for a list of given global variables.

For more information, see "Applying Data Ranges to External Variables and Stub Functions (DRS)".

File format:

The file filename contains a list of global variables with the below format:

```
variable_name val_min val_max <init|permanent|globalassert>
```

Variables scope:

Variables concern external linkage, const variables and not necessary a defined variable (i.e. could be extern with option -allow-undef-variables).

Note Only one mode can be applied to a global variable.

No checks are added with this option except for globalassert mode.

Some warning can be displayed in log file concerning variables when format or type is not in the scope.

Default:

Disable.

Example shell script entry:

```
polyspace-c -data-range-specifications range.txt ...
```

-permissive-stubber

By default, the stubber rejects functions:

- with complex function pointers as parameters
- with function pointers as return type

To eliminate these restrictions and stub all functions, specify the **Stub all functions** (`-permissive-stubber`) option.

Caution Using this option may produce inaccurate results.

Note This option cannot be used with the `no-automatic-stubbing` option.

-no-automatic-stubbing

By default, PolySpace verification automatically stubs all functions. When this option is used, the list of functions to be stubbed is displayed and the verification is stopped.

Benefits:

This option may be used where

- The entire code is to be provided, which may be the case when verifying a large piece of code. When the verification stops, it means the code is not complete.
- Manual stubbing is preferred to improve the selectivity and speed of the verification.

Note This option cannot be used with the `permissive-stubber` option.

Default:

All functions are stubbed automatically

Assumptions

-div-round-down

This option concerns the division and modulus of a negative number.

The ANSI standard stipulates that "*if either operand of / or % is negative, whether the result of the / operator, is the largest integer less or equal than the algebraic quotient or the smallest integer greater or equal than the quotient, is implementation defined, same for the sign of the % operator*".

Note `a = (a / b) * b + a % b` is always true.

Default:

Without the option (default mode), if either operand of */ or %* is negative, the result of the / operator is the smallest integer greater or equal than the algebraic quotient. The result of the % operator is deduced from `a % b = a - (a / b) * b`

Example:

`assert(-5/3 == -1 && -5%3 == -2);` is **true** .

With the *-div-round-down* option:

If either operand */ or %* is negative, the result of the / operator is the largest integer less or equal than the algebraic quotient. The result of the % operator is deduced from `a % b = a - (a / b) * b` .

Example:

`assert(-5/3 == -2 && -5%3 == 1);` is **true** .

Example Shell Script Entry:

polyspace-c -div-round-down ...

-no-def-init-glob

This option specifies that PolySpace verification should not take into account default initialization defined by ANSI C. When this option is not used, default initialization are

- 0 for integers
- 0 for characters
- 0.0 for floats

With the option in use, all global variable will be treated as non initialized - and therefore cause a red error - if they are read before being written to.

Example Shell Script Entry :

```
polyspace-c -no-def-init-glob ...
```

-size-in-bytes

This option allows incomplete or partial allocation of structures. This allocation can be made by malloc or cast .

The example below shows an example using malloc. Further explanation can be found in the section describing the partial and incomplete allocation of structures. Also refer to the -allow-ptr-arith-on-struct section.

```
typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;
BIG *p = malloc(sizeof(LITTLE));
```

Default results

```
 p->a = 0 ;   // red pointer out of its bounds
or p->b = 0 ;  // red pointer out of its bounds
or p->c = 0 ;  // red pointer out of its bounds
```

Results using this option

```
if (p!= ((void *) 0) ) {
 p->a = 0 ;   // green pointer within bounds
or p->b = 0 ;  // green pointer within bounds
or p->c = 0 ;  // red pointer out of its bounds
}
```

-allow-ptr-arith-on-struct

This option enables navigation within a structure or union from one field to another, within the rules defined below. It automatically sets the -size-in-bytes option.

Default

By default, when a pointer points to a variable then the size of the objected pointed to is that of that variable - regardless of whether it is contained within a bigger object, like a structure. Therefore, going out of the scope of this variable leads to a red IDP check (Illegal Dereference Pointer). This is illustrated below.

```
struct S {char a; char b; int c;} x;
 char *ptr = &x.b;
 ptr ++;
 *ptr = 1; // red on the dereference, because the pointed
object was "b"
```

Using this option

When this option is used in the above option, PolySpace verification considers that the object pointed to is now the host object "x". The "ptr" pointer is in fact pointing to &x, with the correct offset to the field “b” within the structure of type S (inter-fields and end-padding included). Therefore, the dereference becomes green

Consider a second example:

```
struct S {
 char a;
 /* 3 bytes of padding between 'a', 'b' */
 int b;
 int c;
 char d[3];
 unsigned char e:7;
 char f;
 /* 3 bytes of end padding */
} x;
char *ptr;
struct Nesting_S {
 struct S s;
 int c;
} z
ptr = (char *)&x.a; ptr++; *ptr = 10; // ptr points to the
padding between a and b
ptr = (char *)&x.b; ptr += 4; *ptr = 10; // ptr points to the
first byte of c
ptr = (char *)&x.d; ptr += 3; *ptr = 10; // ptr points to the
ptr = (char *)&x.f; ptr++; *ptr = 10; // ptr points to the
first byte of end-padding
```

Note For nested structures, for instance with ptr = (char *)&x.d.a, the dereference of *ptr is green if ptr remains within x.d. However, even with this option in use, a red check is generated if the pointer navigates above x.d.a. That is, if this pointer is incremented or decremented such that it now to x.a, x.b, or x.c, it causes a red IDP.

In the third example below, the *ptr access is red regardless of whether the option is set or not.

With the option set, the ptr pointer points to the structure+offset z.s, and ptr can safely navigate within this structure z.s, but z.c is outside it.

Without the option, the ptr pointer points to z.s.f, which is only 1 byte long. So no navigation is allowed, not even within z.s.

```
ptr = (char *)z.s.f; ptr += 4; *ptr = 10; // ptr points to the
first byte of c:
```

-ignore-float-rounding

Without this option, PolySpace verification rounds floats according to the IEEE® 754 standard: simple precision on 32-bits targets and double precision on target which define double as 64-bits.

With the option, **exact** computation is performed.

Example:

```
void ifr(float f)
{
 double a,b;
  a = 0.2;
  b = 0.2;

  if ( a + b == 0.4) {
  // reached whether -ignore-float-rounding is used or not
  assert (1);
  f = 1.0F*f;
  }
 else {
  assert (1);
  f = 1.0F * f;
  // reached only when -ignore-float-rounding is not used
 }
}
```

Using this option can lead to different results compared to the "real life" (compiler and target dependent): Some paths will be reachable or not for PolySpace verification while they are not (or are) depending of the compiler and target. So it can potentially give approximate results (green should be unproven). This option has an impact on OVFL/UNFL checks on floats.

However, this option allows reducing the number of unproven checks because of the “delta” approximation.

For example:

- FLT_MAX (with option set) = 3.40282347e+38F
- FLT_MAX (following IEEE 754 standard) = 3.40282347e+38F ± Δ

```
void ifr(float f)
{
 double a,b;
 a = 0.2;
 b = 0.2;

 if ( a + b == 0.4) {
  assert (1);
  f = 1.0F*f;   // Overflow never occurs because f <= FLT_MAX.
        // reached when -ignore-float-rounding is used
 }
 else {
  assert (1);
  f = 1.0F * f;   // OVFL could occur when f = (FLT_MAX + D)
        // reached when -ignore-float-rounding is not used
 }
}
```

Default:

IEEE 754 rounding under 32 bits and 64 bits.

Example Shell Script Entry:

```
polyspace-c -ignore-float-rounding ...
```

-detect-unsigned-overflows

When this option is selected, verification is more strict with overflowing computations on unsigned integers than the ANSI C standard requires.

The ANSI C standard states that promotion occurs for logic, bitwise and arithmetic operators. For `char`, `short`, and `int` types, variables are implicitly cast into integers before the operation. Then, after the operation, the variables are downcast into the original type.

Consider the examples below.

Example 1

Using this option, the following example generates an error:

```
unsigned char x;
x = 255;
x = x+1;   //overflow due to this option
```

Without this option, however, the example does not generate an error.

```
unsigned char x;
x = 255;
x = x+1;  // turns x into 0 (wrap around)
```

Example 2

Using this option, the following example generates an error:

```
unsigned char Y=1;
Y = ~Y;  //overflow because of type promotion
```

In this example:

1. `Y` is coded as an unsigned char: `000000001`
2. `Y` is promoted to an integer: `00000000 00000000 00000000 00000001`
3. The operation "~" is performed, making `Y`: `11111111 11111111 11111111 11111110`

4 The integer is downcast to an unsigned char, causing an overflow.

Example Shell Script Entry:

```
polyspace-c -detect-unsigned-overflows ...
```

-known-NTC proc1[,proc2[,...]]

After a few verifications, you may discover that a few functions "never terminate". Some functions such as tasks and threads contain infinite loops by design, while functions that exit the program such as *kill_task* , *exit* or *Terminate_Thread* are often stubbed by means of an infinite loop. If these functions are used very often or if the results are for presentation to a third party, it may be desirable to filter all NTC of that kind in the Viewer.

This option is provided to allow that filtering to be applied. All NTC specified at launch will appear in the viewer in the known-NTC category, and filtering will be possible.

Default :

All checks for deliberate Non Terminating Calls appear as red errors, listed in the same category as any problem NTC checks.

Example Shell Script Entry :

```
polyspace-c -known-NTC "kill_task,exit"

polyspace-c -known-NTC "Exit,Terminate_Thread"
```

Automatic Orange Tester

-prepare-automatic-tests

This option activates the PolySpace Automatic Orange Tester. The Automatic Orange Tester finds runtime errors in the orange (and red) checks remaining at the end of the PolySpace verification.

The Automatic Orange Tester results contain precise information to help you identify the cause of a runtime error. This complements the results review in the Viewer module of PolySpace Client for C/C++.

For more information, see “Automatically Testing Orange Code”.

The following options are not compatible with `-prepare-automatic-tests`.

- `-entry-points`
- `-dialect`
- `-ignore-float-rounding`
- `-div-round-down`
- `-entry-points`
- `-char-is-16bits`
- `-short-is-8bits`
- `-respect-types-in-globals`
- `-respect-types-in-fields`

The following options cannot take specific values when you select -prepare-automatic-tests.

- `-align [8|16]`
- `-target [c-167 | tms320c3c | hc08 | sharc21x61]`
- `-data-range-specification` (in global assert mode)

In addition, when using the Automatic Orange Tester, the `-target mcpu` option must be used together with `-pointer-is-32bits`.

Default :

Disabled

Example Shell Script Entry :

```
polyspace-c -prepare-automatic-tests ...
```

-machine-architecture

This option specifies whether verification runs in 32 or 64-bit mode.

Note You should only use the option `-machine-architecture 64` for verifications that fail due to insufficient memory in 32 bit mode. Otherwise, you should always run in 32–bit mode.

Available options are:

- `-machine-architecture auto` – Verification always runs in 32-bit mode.
- `-machine-architecture 32` – Verification always runs in 32-bit mode.
- `-machine-architecture 64` – Verification always runs in 64-bit mode.

Default:

```
auto
```

Example Shell Script Entry:

```
polyspace-c -machine-architecture auto
```

-max-processes

This option specifies the maximum number of processes that can run simultaneously on a multi-core system. The valid range is 1 to 128.

Note To disable parallel processing, set: `-max-processes 1`.

Default:

```
4
```

Example Shell Script Entry:

```
polyspace-c -max-processes 1
```

Others

-extra-flags option-extra-flag

This option specifies an expert option to be added to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

These flags will be given to you by Technical Support as necessary for your verifications.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -extra-flags -param1 -extra-flags -param2 \

 -extra-flags 10 ...
```

-c-extra-flags flag

This option is used to specify an expert option to be added to a verification. Each word of the option (even the parameters) must be preceded by *-c-extra-flags*.

These flags will be given to you by PolySpace as necessary for your verifications.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -c-extra-flags -param1 -c-extra-flags -param2
-c-extra-flags 10
```

-il-extra-flags flag

This option is used to specify an expert option to be added to a verification. Each word of the option (even the parameters) must be preceded by *-il-extra-flags*.

These flags will be given to you by PolySpace as necessary for your verifications.

Default:

No extra flags.

Example Shell Script Entry:

```
polyspace-c -il-extra-flags -param1 -il-extra-flags -param2
-il-extra-flags 10
```

Precision/Scaling Options

In this section...

-quick (Deprecated)

Note This option is deprecated in R2009a and later releases.

`quick` mode is obsolete and has been replaced with verification PASS0. PASS0 takes somewhat longer to run, but the results are more complete. The limitations of `quick` mode, (no NTL or NTC checks, no float checks, no variable dictionary) no longer apply. Unlike `quick` mode, PASS0 also provides full navigation in the Viewer.

This option is used to select a very fast mode for PolySpace .

Benefits

This option allows results to be generated very quickly. These are suitable for initial verification of red and gray errors only, as orange checks are too plentiful to be relevant using this option.

Limitations

- No NTL or NTC are displayed (non termination of loop/call)
- The variable dictionary is not available
- No check is performed on floats
- The call tree is available but navigation is not possible
- Orange checks are too plentiful to be relevant

-O(0-3)

This option specifies the precision level to be used. It provides higher selectivity in exchange for more verification time, therefore making results review more efficient and hence making bugs in the code easier to isolate. It does so by specifying the algorithms used to model the program state space during verification.

The MathWorks recommends you begin with the lowest precision level. Red errors and gray code can then be addressed before relaunching PolySpace verification using higher precision levels.

Benefits:

- A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate.
- A higher precision level also means higher verification time
  - -O0 corresponds to static interval verification.
  - -O1 corresponds to complex polyhedron model of domain values.
  - -O2 corresponds to more complex algorithms to closely model domain values (a mixed approach with integer lattices and complex polyhedrons).
  - -O3 is only suitable for code smaller than 1000 lines of code. For such codes, the resulting selectivity might reach high values such as 98%, resulting in a very long verification time, such as an hour per 1000 lines of code.

Default:

-O2

Example Shell Script Entry:

```
polyspace-c -O1 -to pass4 ...
```

-modules-precision mod1:O(0-3)[,mod2:O(0-3)[,...]]

This option is used to specify the list of .c files to be analyzed with a different precision from that specified generally -O(0-3) for this verification.

In batch mode, each specified module is followed by a colon and the desired precision level for it. Any number of modules can be specified in this way, to form a comma-separated list with no spaces.

Default:

All modules are treated with the same precision.

Example Shell Script Entry:

```
polyspace-c -O1 \
-modules-precision myMath:O2,myText:O1, ...
```

-from verification-phase

This option specifies the verification phase to start from. It can only be used on an existing verification, possibly to elaborate on the results that you have already obtained.

For example, if a verification has been completed -to pass1, PolySpace verification can be restarted *-from* pass1 and hence save on verification time.

The option is usually used in a verification after one run with the -to option, although it can also be used to recover after power failure.

Possible values are as described in the -to *verification-phase* section, with the addition of the *scratch* option.

Note

- This option can only be used for client verifications. All server verifications start from *scratch*.
- Unless the *scratch* option is used, this option can be used only if the previous verification was launched using the option *-keep-all-files* .
- This option cannot be used if you modify the source code between verifications.

Default :

scratch

Example Shell Script Entry :

```
polyspace-c -from c-to-il ...
```

-to verification-phase

This option specifies the phase after which the verification will stop.

Benefits:

This option provides improved selectivity, making results review more efficient and making bugs in the code easier to isolate.

- A higher integration level contributes to a higher selectivity rate, leading to "finding more bugs" with the code.
- A higher integration level also means higher verification time

---

Note The MathWorks recommends you begin by running `-to pass0` (`Software Safety Analysis level 0`) You can then address red errors and gray code before relaunching verification using higher integration levels.

Possible values:

- `c-compile` or `"C Source Compliance Checking"`
- `c-to-il` or `"C to Intermediate Language"`
- `pass0` or “`Software Safety Analysis level 0`”
- `pass1` or `"Software Safety Analysis level 1"`
- `pass2` or `"Software Safety Analysis level 2"`
- `pass3` or `"Software Safety Analysis level 3"`
- `pass4` or `"Software Safety Analysis level 4"`
- `other`

Note If you use *-to other* then PolySpace verification will continue until you stop it manually (via `kill-rte-kernel`) or stops until it has reached *pass20*.

Default:

pass4

Example Shell Script Entry:

```
polyspace-c -to "Software Safety Analysis level 3"...

polyspace-c -to pass0 ...
```

-context-sensitivity "proc1[,proc2[,...]]"

This option allows the precise verification of a procedure with regards to the discrete calls to it in the analyzed code.

Each check inside the procedure is split into several sub-checks depending on the context of call. Therefore if a check is red for one call to the procedure and green for another, both colors will be revealed.

This option is especially useful if a problem function is called from a multitude of places.

-context-sensitivity-auto

This option is similar to the -context-sensitivity option, except that the system automatically chooses the procedures to be considered.

-path-sensitivity-delta number

This option is used to improve interprocedural verification precision within a particular pass (see -to *pass1*, *pass2*, *pass3* or *pass4*). The propagation of information within procedures is done earlier than usual when this option is specified. That results in improved selectivity and a longer verification time.

Consider two verifications, one with this option set to 1 (with), and one without this option (without)

- a level 1 verification in (with) (pass1) will provide results equivalent to level 1 or 2 in the (without) verification
- a level 1 verification in (with) can last x times more than a cumulated level 1+2 verification from (without). "x" might be exponential.

- the same applies to level 2 in (with) equivalent to level 3 or 4 in (without), with potentially exponential verification time for (a)

Gains using the option

- (+) highest selectivity obtained in level 2. no need to wait until level 4
- (-) This parameter increases exponentially the verification time and might be even bigger than a cumulated verification in level 1+2+3+4
- (-) This option can only be used with less than 1000 lines of code

Default:

0

Example Shell Script Entry:

```
polyspace-c -path-sensitivity-delta 1 ...
```

-retype-pointer

This option can be used to retype variables of pointer types in order to improve precision of pointer conversions chain.

The principle consists in replacing original type by the aliased object type when a symbol of pointer type aliases to a single type of objects.

For example, following assert can be proved using -retype-pointer option:

```
struct A {int a; char b;} s = {1,2};
char *tmp = (char *)&s;
struct A *pa = (struct A*)tmp;
assert((pa->a == 1) && (pa->b == 2));
```

This principle can be applied to fields of struct/unions of a pointer type. However, this option set -size-in-bytes option and it does not have expected precision with -allow-ptr-arith-on-struct.

Moreover, this option is forbidden when using -retype-int-pointer option.

Default:

disable by default

Example Shell ScriptEntry:

```
polyspace-c -retype-pointer ...
```

-retype-int-pointer

This option can be used to retype variables of pointer to signed or unsigned integer types in order to improve precision of pointer conversions chain.

The principle consists in replacing original type by the aliased object type when a symbol of pointer type aliases to a single type of objects. It applies only on symbols of signed or unsigned integer types.

For example, following assert can be proved using -retype-int-pointer option:

```
void function(void)
 {
  struct S1 {
   int x;
   int y;
   int z;
   char t;
  } s1 = {1,2,3,4};
 struct S2 {
  int first;
  void *p;
 } s2;
 int addr;
 addr = (int)(&s1);
 assert(((struct S1 *)addr)->y == 2); // ASRT is verified
 s2.first = (int)(&s1);
 assert(((struct S1 *)s2.first)->y == 2); // ASRT is verified
}
```

However, this option set -size-in-bytes and has no effect when set -respect-types-in-globals on global symbols of integer types and when set -respect-types-in-fields on fields of struct/union of integer types.

Some sides effects can be noticed on PolySpace checks concerning initialization on variables which can be stated as initialization on pointer check (NIP).

This option requires the -retype-pointer option.

This option should be used on:

- **Code with memory mapping** – When constant bg structures (global variable) are declared with a pointer and points to const structure, setting the option will consider that the pointer and the pointer structure are synonyms (aliased) and precision of the result will increase. Option to set: `-retype-pointer`.
- **Code close to the communication layer API** (code with lot of cast in (void *)) – When code contains low level drivers, generic pointer (void *) can be used. It is recommended to use this with an `-inline` of the functions containing these casts. Options to set: `-retype-pointer -inline`.
- **Code in which MISRA rule 11.2 is violated** – When integers contains pointers, precision can be improved when setting an option. Option to set: `-retype-int-pointer`.

These options are not set by default because they all change the option `-size-in-bytes`. Therefore, precision can reduced and some red IDP checks may be affected. In addition, using these options will consider "x" (previously int) as a pointer. This results in checks changing category (NIV to NIP).

Default:

Disable by default

Example Shell ScriptEntry:

```
polyspace-c -retype-int-pointer...
```

-k-limiting number

This is a scaling option to limits the depth of verification into nested structures during pointer verification.

This option is only available for C and C++.

Default:

There is no fixed limit.

Example Shell Script Entry:

```
polyspace-c -k-limiting 1 ...
```

In this example above, verification will be precise to only one level of nesting.

-no-fold

When variables are defined with huge static initialization, scaling problems may occur during the compilation phase. This option approximates the initialization of array types of integer, floating point, and char types (included string) if needed.

It can speed up the verification, but may decrease precision for some applications

Default:

Option not set.

Example Shell Script Entry:

```
polyspace-c -no-fold ...
```

-respect-types-in-globals

This is a scaling option, designed to help process complex code. When it is applied, PolySpace verification assumes that global variables not declared as containing pointers are never used for holding pointer values. This option

should only be used with Type-safe code, when it does not cause a loss of precision. See also -respect-types-in-fields.

In the following example, we will lose precision using option –respect-types-in-globals option:

```
int x;
void t1(void) {
 int y;
 int *tmp = &x;
 *tmp = (int)&y;
 y=0;
 *(int*)x = 1; // x contains address of y
 assert (y == 0); // green with the option
}
```

PolySpace verification will not take care that x contains the address of y resulting a green assert.

Default:

PolySpace verification assumes that global variables may contain pointer values.

Example Shell Script Entry:

```
polyspace-c -respect-types-in-globals ...
```

-respect-types-in-fields

This is a scaling option, designed to help process complex code. When it is applied, PolySpace verification assumes that structure fields not declared as containing pointers are never used for holding pointer values. This option should only be used with Type-safe code, when it does not cause a loss of precision. See also -respect-types-in-globals.

In the following example, we will lose precision using option –respect-types-in-fields option:

```
struct {
 unsigned x;
```

```
 int f1;
 int *z[2];
} S1;

void funct2(void) {
 int *tmp;
 int y;
 ((int**)&S1)[0] = &y;  /* S1.x points on y */
 tmp = (int*)S1.x;
 y=0;
 *tmp = 1;      /* write 1 into y */
 assert(y==0);
}
```

PolySpace verification will not take care that S1.x contains the address of y resulting a green assert.

Default:

PolySpace verification assumes that structure fields may contain pointer values.

Example Shell Script Entry:

```
polyspace-c -respect-types-in-fields ...
```

-inline "proc1[,proc2[,...]]"

A scaling option that creates a clone of a each specified procedure for each call to it.

Cloned procedures follow a naming convention viz:

```
procedure1_pst_cloned_nb,
```

where nb is a unique number giving the total number of cloned procedures.

Such an inlining allows the number of aliases in a given procedure to be reduced, and may also improve precision.

Restrictions :

- Extensive use of this option may duplicate too much code and may lead to other scaling problems. Carefully choose procedures to inline.
- This option should be used in response to the inlining hints provided by the alias verification
- This option should not be used on main, task entry points and critical section entry points

-lightweight-thread-model

This scaling option can be used to reduce task complexity (see also -entry-points).

It uses a slightly less precise model of pointer/thread interaction compared to that used by default, and is likely to prove helpful when there are a lot of pointers in an application. See Chapter 3, “Approximations Used During Verification” for more explanation of when to use it.

It causes a loss of precision:

- It causes a slight loss of precision when shared variables are reads via pointers.
- Some read/write accesses may not appear in the Global Variable Dictionary.

Default:

disabled by default.

Example Shell Script Entry :

```
polyspace-c -lightweight-thread-model ...
polyspace-c -lwtm ...
```

-less-range-information

Limits the amount of range information displayed in verification results.

When you select this option, range information is provided on assignments, but not on reads and operators.

Scaling problems can occur when computing range information on reads and operators. In these cases, using this option can significantly increase the speed of the verification.

Default:

Disabled.

Example Shell Script Entry :

```
polyspace-c -less-range-information
```

Multitasking Options (PolySpace Server for C/C++ Product Only)

In this section...

Note Concurrency options are not compatible with -main-generator options.

-entry-points str1[,str2[,...]]

This option is used to specify the tasks/entry points to be analyzed by PolySpace server, using a Comma-separated list with no spaces.

These entry points must not take parameters. If the task entry points are functions with parameters they should be encapsulated in functions with no parameters, with parameters passed through global variables instead.

Using PolySpace verification, c tasks must have the prototype "void *task_name*(void);".

Example Shell Script Entry:

```
polyspace-c -entry-points proc1,proc2,proc3 ...
```

-critical-section-[begin or end] "proc1:cs1[,proc2:cs2]"

```
-critical-section-begin "proc1:cs1[,proc2:cs2]"
```

and

```
-critical-section-end "proc3:cs1[,proc4:cs2]"
```

These options specify the procedures beginning and ending critical sections, respectively. Each uses a list enclosed within double speech marks, with

list entries separated by commas, and no spaces. Entries in the lists take the form of the procedure name followed by the name of the critical section, with a colon separating them.

These critical sections can be used to model protection of shared resources, or to model interruption enabling and disabling.

Note This option cannot be used for client verifications, or with the `main-generator` option.

Default:

no critical sections.

Example Shell Script Entry:

```
polyspace-c -critical-section-begin "start_my_semaphore:cs" \

-critical-section-end "end_my_semaphore:cs"
```

-temporal-exclusions-file file_name

This option specifies the name of a file. That file lists the sets of tasks which never execute at the same time (temporal exclusion).

The format of this file is :

- one line for each group of temporally excluded tasks,
- on each line, tasks are separated by spaces.

Note This option cannot be used for client verifications, or with the `main-generator` option.

Default:

No temporal exclusions.

Example Task Specification file

File named 'exclusions' (say) in the 'sources' directory and containing:

```
task1_group1 task2_group1

task1_group2 task2_group2 task3_group2
```

Example Shell Script Entry :

```
polyspace-c -temporal-exclusions-file sources/exclusions \
    -entry-points task1_group1,task2_group1,task1_group2,\
    task2_group2,task3_group2 ...
```

Batch Options

In this section...

-server server_name_or_ip[:port_number]

Using polyspace-remote-[c] [–server [name or IP address][:<port number>]] allows you to send a verification to a specific or referenced PolySpace server.

Note If the option –server is not specified, the default server referenced in the PolySpace-Launcher.prf configuration file will be used as the server.

When a –server option is associated to the batch launching command, the name or IP address and a port number need to be specified. If the port number does not exist, the 12427 value will be used by default.

Note also that polyspace-remote- accepts all other options.

Option Example Shell Script Entry:

```
polyspace-remote-c  server 192.168.1.124:12400

polyspace-remote-c

polyspace-remote-c  server Bergeron
```

-sources-list-file file_name

This option is only available in batch mode. The syntax of *file_name* is the following:

- One file per line.
- Each file name includes its absolute or relative path.

The source files are compiled in the order in which they are specified.

Note If you do not specify any files, the software verifies all files in the source directory in alphabetical order.

Example Shell Script Entry for -sources-list-file:

```
polyspace-c -sources-list-file "C:\Analysis\files.txt"

polyspace-c -sources-list-file "files.txt"
```

-v | -version

Display the PolySpace version number.

Example Shell Script Entry:

```
polyspace-c  v
```

It will show a result similar to:

```
PolySpace r2007a+

Copyright (c) 1999-2008 The Mathworks, Inc.
```

-h[elp]

Display in the shell window a simple help in a textual format giving information on all options.

Example Shell Script Entry:

```
polyspace-c  h
```

2

Check Descriptions

Colored Source Code for C

In this section...

Illegal Pointer Access to Variable or Structure Field: IDP

This is a check to establish whether in the dereferencing of an expression of the form *ptr+i*, the variable/structure field initially pointed to by *ptr*is still the one accessed. See ANSI C standard ISO/IEC 9899 section 6.3.6.

Consider the following example.

```
1 int a;
2
3 struct {
4  int f1;
5  int f2;
6  int f3;
7 } S;
8
9 void main(void)
10 {
11  volatile int x;
12
13  if (x)
14   *(&a+1) = 2;
15  // IDP ERROR: &a +1 doesn't point to a any longer
16  if (x)
17   *(&S.f1 +1) = 2;
18  // IDP ERROR: you are not allowed to access f2 like this
19 }
```

According to the ANSI C standard, it is not permissible to access a variable (or a structure field) from a pointer to another variable. That is, *ptr+i* may only be dereferenced if *ptr+i* is the address of a subpart of the object pointed to by *ptr*(such as an element of the array pointed to by *ptr*, or a field of the structure pointed to by *ptr*).

For instance, the following code is correct because the length of the entity pointed to by *ptr_s*reflects the full structure length of *My_struct* (at line 11):

```
1 typedef struct {
2  int f1;
3  int f2;
```

```
4  int f3;
5 } My_Struct;
6
7 My_Struct s = {1,2,3};
8
9 int main(void)
10 {
11 My_Struct *ptr_s = &s;
12
13  // change to f2 field
14  *((int *)&s +1) = 2; // Correct evaluation
15
16  return 0;
17 }
```

Array Conversion Must Not Extend Range: COR

This is a check to establish whether a small array is mapped onto a bigger one through a pointer cast. Consider the following example.

```
1
2 typedef int Big[100];
3 typedef int Small[10];
4 typedef short EquivBig[200];
5
6 Small smalltab;
7 Big bigtab;
8
9 void main(void)
10 {
11  volatile int random;
12
13  Big * ptr_big = &bigtab;
14  Small * ptr_small = &smalltab;
15
16  if (random) {
17   Big *new_ptr_big = (Big*)ptr_small; // COR ERROR: array
conversion must not extend range
18  }
19  if (random) {
20   EquivBig *ptr_equivbig = (EquivBig*)ptr_big;
```

```
21   Small *ptr_new_small = (Small*)ptr_big; // Conversion
verified
22  }
23 }
```

In the example above, a pointer is initialized to the *Big* array with the address of the *Small* array. This is not legal since it would be possible to dereference this pointer outside the *Small* array. Line 20 shows that the mapping of arrays with same size and different prototypes is acceptable.

Array Index Within Bounds: OBAI

This is a check to establish whether an index accessing an array is compatible with the length of that array. Consider the following example.

```
1
2 #define TAILLE_TAB 1024
3 int tab[TAILLE_TAB];
4
5 void main(void)
6 {
7  int index;
8
9  for (index = 0; index < TAILLE_TAB ; index++)
10   {
11    tab[index] = 0;
12   }
13  tab[index] = 1;
14 // OBAI ERROR: Array index out of bounds [0..1023]
15 }
```

Just after the loop, *index* equals *SIZE_TAB*. Thus *tab[index] = 1* overwrites the memory cell just after the last array element.

An OBAI check can also be localized on a + operator, as another example illustrates.

```
1 int tab[10];
2
3 void main(void)
4 {
```

```
5   int index;
6  for (index = 0; index < 10 ; index++)
7   *(tab + index) = 0;
8
9  *(tab + index) = 1; // OBAI ERROR: Array index out of bounds
10 }
```

Note that the message associated with the check OBAI gives always the range of the array: Array index out of bounds [0..1023]

Initialized Return Value: IRV

This is a check to establish whether a function returns an initialized value. Consider the following example.

```
1
2 extern int random_int(void);
3
4 int reply(int msg)
5 {
6  int rep = 0;
7  if (msg > 0) return rep;
8 }
9
10 void main(void)
11 {
12  int ans;
13
14  if (random_int())
15   ans = reply(1); // IRV verified: function returns an
initialised value
16  else if (random_int())
17   ans = reply(0); // IRV ERROR: function does not return an
initialised value
18  else
19   reply(0); // No IRV checks because the return value
is not used
20
21 }
22
```

```
23
```

Variables are often initialized using the return value of functions. However, in the above example the return value is not initialized for all input parameter values. In this case, the target variable will not be always be properly initialized with a valid return value.

Non-Initialized Variables: NIV/NIVL

This is a check to establish whether a variable is initialized before being read. Consider the following example.

```
1
2 extern int random_int(void);
3
4 void main(void)
5 {
6  int x,i;
7  double twentyFloat[20];
8  int y = 0;
9
10  if (random_int()) {
11   y += x; // NIV ERROR: Non
Initialized Variable (type: int 32)
12  }
13  if (random_int()) {
14   for (i = 1; i < 20; i++) {
15    if (i % 2) twentyFloat[i] = 0.0;
16   }
17   twentyFloat[2] = twentyFloat[4] + 5.0; // NIV Warning.
Only odd indexes are initialized.
18  }
19 }
```

The result of the addition is unknown at line 11 because *x* is not initialized (UNR unreachable code on "+" operator).

In addition, line 17 shows how PolySpace software prompts the user to investigate further (by means of an orange check) when all cells have not been initialized.

Note Associated to each message which concerns a NIV check, PolySpace software gives the type of the variable like the following examples: (type: volatile int32), (type: int 16), (type: unsigned int 8), etc.

Non-Initialized Pointer: NIP

Check to establish whether a reference is initialized before being dereferenced. Consider the following example.

```
2
3 void main(void)
4 {
5  int* p;
6  *p = 0; // NIP ERROR: reference is not initialized
7 }
```

As *p* is not initialized, an undefined memory cell would be overwritten at line 6 (**p = 0*) (also leading to the unreachable gray check on "*").

POW (Deprecated)

Note The POW check is deprecated in R2009a and later. The POW check no longer appears in PolySpace results.

Check to establish whether the standard **pow** function from *math.h* library is used with an acceptable (positive) argument.

User Assertion: ASRT

This is a check to establish whether a user assertion is valid. If the assumption implied by an assertion is invalid, then the standard behavior of the assert macro is to abort the program. PolySpace verification therefore considers a failed assertion to be a runtime error. Consider the following example.

```
1 #include <assert.h>
2
3 typedef enum
```

```
4 {
5  monday=1, tuesday,
6  wensday, thursday,
7  friday, saturday,
8  sunday
9 } dayofweek ;
10
11 // stubbed function
12 dayofweek random_day(void);
13 int random_value(void);
14
15 void main(void)
16 {
17  unsigned int var_flip;
18  unsigned int flip_flop;
19  dayofweek curDay;
20  unsigned int constant = 1;
21
22  if (random_value()) flip_flop=1; else flip_flop=0;
 // flip_flop can randomly be 1 or 0
23  var_flip = (constant | random_value());
// var_flip is always > 0
24
25  if(random_value()) {
26   assert(flip_flop==0 || flip_flop==1); // User Assertion is
verified
27   assert(var_flip>0); // User Assertion is
verified
28   assert(var_flip==0); // ASRT ERROR: Failure User
Assert
29  }
30
31  if (random_value()) {
32   curDay = random_day(); // Random day of the week
33   assert( curDay > thursday); // ASRT Warning: User
assertion may fails
34   assert( curDay > thursday); // User assertion is
verified
35   assert( curDay <= thursday); // ASRT ERROR: Failure User
Assertion
```

```
36  }
37 }
```

In the *main*, the *assert* function is used in two different manners:

1. To establish whether the values *flip_flop* and *var_flip* in the program are inside the domain which the program is designed to handle. If the values were outside the range implied by the *assert* (see line 28), then the program would not be able to run properly. Thus they are flagged as runtime errors.

2. To redefine the range of variables as shown at line 34 where *curDay* is restricted to just a few days. Indeed, PolySpace verification makes the assumption that if the program is executed without a runtime error at line 33, *curDay* can only have a value greater than *thursday* after this line.

Scalar and Float Underflows: UNFL

These are checks to establish whether arithmetic expressions underflow. A scalar check is used with integer type, and a float check for floating point expressions. Consider the following example.

```
1 #include <float.h>
2 extern int random_int(void);
3
4 void main(void)
5 {
6  int i = 1;
7  float fval = FLT_MAX;
8
9  i = -2 * (i << 30); // i = -2**31
10  if (random_int()) i = i - 1; // UNFL ERROR: scalar
variable is underflow
11  if (random_int()) fval = -2 * fval; // UNFL ERROR: float
variable is underflow
12 }
13
```

The minimum integer value on a 32 bit architecture platform is represented by *-2**31*, thus adding *(-1)*will raise an underflow.

Float Underflow Versus Values Near Zero: UNFL

The definition of the word "underflow" differs between the ANSI standard and the ANSI/IEEE 754-1985 standard. According to the former definition, underflow occurs when a number is sufficiently negative for its type not to be capable of representing it. According to the latter, underflow describes the erroneous representation of a value close to zero due to the limits of its representation.

PolySpace verifications apply the former definition. The latter definition does not impose the raising of an exception as a result of an underflow. By default, processors supporting this standard permit the deactivation of such exceptions.

Consider the following example.

```
2 #define FLT_MAX 3.40282347e+38F // maximum representable
float found in <float.h>
3 #define FLT_MIN 1.17549435e-38F // minimum normalised
float found in <float.h>
4
5 void main(void)
6 {
7  float zer_float = FLT_MIN;
8  float min_float = -(FLT_MAX);
9
10  zer_float = zer_float * zer_float; // No check underflow
near zero
11  min_float = min_float * min_float; // UNFL ERROR:
underflow checked by verifier
12
13 }
```

Scalar and Float Overflows: OVFL

These are checks to establish whether arithmetic expressions overflow. This is a scalar check with integer type and float check for floating point expression. Consider the following example.

```
1 #include <float.h>
2 extern int random_int(void);
3
```

```
4 void main(void)
5 {
6  int i = 1;
7  float fvalue = FLT_MAX;
8
9  i = i << 30; // i = 2**30
10  if (random_int())
11   i = 2 * (i - 1) + 2; // OVFL ERROR: 2**31 is an overflow
value for int32
12  if (random_int())
13   fvalue = 2 * fvalue + 1.0; // OVFL ERROR: float variable is
overflow
14 }
```

On a 32 bit architecture platform, the maximum integer value is *2**31-1*, thus *2**31*will raise an overflow.

In the same manner, if *fvalue*represents the biggest float its double cannot be represented with same type and raises an overflow.

How Much is the Biggest Float in C?

There are occasions when it is important to understand when overflow may occur on a float value approaching its maximum value. Consider the following example.

```
void main(void)
{
 float x, y;
 x = 3.40282347e+38f; // is green
 y = (float) 3.40282347e+38; // OVFL red
}
```

There is a red error on the second assignment, but not the first. The real "biggest" value for a float is: `340282346638528859811704183484516925440.0 - MAXFLOAT -.`

Now, rounding is not the same when casting a constant to a float, or a constant to a double:

- floats are rounded to the nearest lower value;

- doubles are rounded to the nearest higher value;
- 3.40282347e+38 is strictly bigger than 340282346638528859811704183484516925440 (named MAXFLOAT).
- In the case of the second assignment, the value is cast to a double first - by your compiler, using a temporary variable D1 -, then into a float - another temporary variable -, because of the cast. Float value is greater than MAXFLOAT, so the check is red.
- In the case of the first assignment, 3.40282347e+38f is directly cast into a float, which is less than MAXFLOAT

The solution to this problem is to use the "f" suffix to specify the variable directly as a float, rather than casting.

What is the Type of Constants/What is a Constant Overflow?

Consider the following example, which would cause an overflow.

```
int x = 0xFFFF; /* OVFL */
```

The type given to a constant is the first type which can accommodate its value, from the appropriate sequence shown below. (See “Predefined Target Processor Specifications (size of char, int, float, double...)” in the *PolySpace Products for C User’s Guide* for information about the size of a type depending on the target.)

Decimals	int , long , unsigned long
Hexadecimals	Int, unsigned int, long, unsigned long
Floats	double

For examples (assuming 16-bits target):

5.8	double
6	int
65536	long
0x6	int

0xFFFF	unsigned int
5.8F	float
65536U	unsigned int

The options -ignore-constant-overflows allow the user to bypass this limitation and consider the line:

int x = 0xFFFF; /* OVFL */ as int x = -1; instead of 65535, which does not fit into a 16-bit integer (from -32768 to 32767).

Left shift overflow on signed variables: OVFL

Overflows can be also be encountered in the case of left shifts on signed variables. In the following example, the higher order bit of *0x41021011* (hexadecimal value of *1090654225*) has been lost, highlighting an overflow (integer promotion).

```
1
2 void main(void)
3 {
4  int i;
5
6  i = 1090654225 << 1; // OVFL ERROR: on left shift range
7 }
```

Float Underflows and Overflows: UOVFL (Deprecated)

Note The UOVFL check is deprecated in R2009a and later. The UOVFL check no longer appears in PolySpace results. Instead of a single UOVFL check, the results now display two checks, a UNFL and an OVFL.

The check UOVFL only concerns float variables. PolySpace verification shows an UOVFL when both overflow and underflow can occur on the same operation.

```
1 #include <math.h>
2 extern int random(void);
```

```
3 #define FLT_MAX 3.40282347e+38F
4
5 int toto(void)
6 {
7  float x;
8  if (random())
9  {
10   x = -FLT_MAX;
11  }
12  else if (random())
13  {
14   x = FLT_MAX;
15  }
16  else
17  {
18   x = 0;
19  }
20  x = 2.0F * x; // UOVFL unproven: possible overflow and
underflow
21  return 1;
22 }
```

According to the branch in use, the results of the operation 2.0F * x could overflow or underflow.

Scalar or Float Division by Zero: ZDV

This is a check to establish whether the right operand of a division (that is, the denominator) is different from 0[.0]. Consider the following example.

```
1 extern int random_value(void);
2
3 void zdvs(int p)
4 {
5  int i, j = 1;
6  i = 1024 / (j-p); // ZDV ERROR: Scalar Division by Zero
7 }
8
9 void zdvf(float p)
10 {
```

```
11  float i,j = 1.0;
12  i = 1024.0 / (j-p); // ZDV ERROR: float Division by Zero
13 }
14
15 int main(void)
16 {
17  volatile int random;
18  if (random_value()) zdvs(1);
// NTC ERROR: because of ZDV ERROR
in ZDVS.
19  if (random_value()) zdvf(1.0);
// NTC ERROR: because of ZDV ERROR
in ZDVF.
20 }
```

Shift Amount in 0..31 (0..63):SHF

This is a check to establish whether a shift (left or right) is bigger than the size of the integral type operated upon (int or long int). The range of allowed shift depends on the target processor: 16 bits on *c-167*, 32 bits on *i386* for int, etc. Consider the following example.

```
1 extern int random_value(void);
2
3 void main(void)
4 {
5  volatile int x;
6  int k, l = 1024; // 32 bits on i386
7  unsigned int v, u = 1024;
8
9   if (x) k = l << 16;
10  if (x) k = l >> 16;
11
12  if (x) k = l << 32; // SHF ERROR
13  if (x) k = l >> 32; // SHF ERROR
14
15  if (x) v = u >> 32; // SHF ERROR
16  if (x) k = u << 32; // SHF ERROR
17
18 }
```

In this example, it is shown that the shift amount is greater than the integer size.

Left Operand of Left Shift is Negative: SHF

This is a check to establish whether the operand of a left shift is a signed number. Consider the following example.

```
1
2
3 void main(void)
4 {
5  int x = -200;
6  int y;
7
8  y = x << 1; // SHF ERROR: left operand must be positive
9
10 }
```

As an aside, note that the -allow-negative-operand-in-shift option used at launching time instructs PolySpace software to permit explicitly signed numbers on shift operations. Using the option in the example above would see the red check at line 8 transformed in a green one. Similarly, if the verification had included the expression *-2 << 2* at line 9, then that line would have been given a green check and y would assume a values of *-8*.

Function Pointer Must Point to a Valid Function: COR

This is a check to establish whether a function pointer points to a valid function, or to function with a valid prototype. Consider the following example.

```
1
2 typedef void (*CallBack)(float *data);
3
4 struct {
5  int ID;
6  char name[20];
7  CallBack func;
8 } funcS;
9
10 float fval;
```

```
11
12 void main(void)
13 {
14  CallBack cb = (CallBack)((char*)&funcS + 24*sizeof(char));
15
16  cb(&fval); // COR ERROR: function pointer must point
to a valid function
17 }
```

In the example above, *func* has a prototype in conformance with *CallBack*'s declaration. Therefore *func* is initialized to point to the NULL function through the global declaration of *funcS*.

Consider a second example.

```
1
2 #define MAX_MEMSEG 32764
3 typedef void (*ptrFunc)(int memseg);
4 ptrFunc initFlash = (ptrFunc)(0x003c);
5
6 void main(void)
7 {
8  int i;
9
10  for (i = 0 ; i < MAX_MEMSEG; i++) // NTL propagation
11  {
12   initFlash(i); // COR ERROR: function pointer must point to a
valid function
13  }
14
15 }
```

As PolySpace verification does not take the memory mapping of programs into account, it cannot ascertain whether *0x003* is the address of a function code segment or not (for instance, as far as PolySpace verification is concerned it could be a data segment). Thus a certain (red) error is raised.

Wrong Type for Argument: COR

This is a check to establish whether each argument passed to a function matches the prototype of that function. Consider the flowing example.

```
1
2 typedef struct {
3  float r;
4  float i;
5 } complex;
6
7 typedef int (*t_func)(complex*);
8
9 int foo_type(int *x)
10 {
11  if (*x%2 == 0) return 0;
12  else return 1;
13 }
14
15 void main(void)
16 {
17  t_func ptr_func;
18  int j,i = 0;
19
20  ptr_func = foo_type;
21  j = ptr_func(&i); // COR ERROR: wrong type of argument for #1
22 }
23
```

In this example, *ptr_func* is a pointer to a function which expects a pointer to a *complex* as input argument. However, the parameter used is a pointer to an *int*.

Wrong Number of Arguments: COR

This is a check to establish whether the number of arguments passed to a function matches the number of arguments in its prototype. Consider the following example.

```
1
2 typedef int (*t_func_2)(int);
```

```
3 typedef int (*t_func_2b)(int,int);
4
5 int foo_nb(int x)
6 {
7  if (x%2 == 0)
8   return 0;
9  else
10   return 1;
11 }
12
13
14 void main(void)
15 {
16  t_func_2b ptr_func;
17  int i = 0;
18
19  ptr_func = (t_func_2b)foo_nb;
20  i = ptr_func(1,2); // COR ERROR: the wrong number of arguments
21 }
22
```

In this example, *ptr_func*is a pointer to a function that takes two arguments but it has been initialized to point to a function that only takes one.

Wrong Return Type of a Function Pointer: COR

This is a check to establish whether the return type passed to a function pointer matches the declaration in its prototype. Consider the following example.

```
1
2 typedef int (*t_func_2)(int);
3 typedef double (*t_func_2b)(int);
4
5 int foo_nb(int x)
6 {
7  if (x%2 == 0)
8   return 0;
9  else
10   return 1;
```

```
11 }
12
13
14 void main(void)
15 {
16  t_func_2b ptr_func;
17  int i = 0;
18
19  ptr_func = (t_func_2b)foo_nb;
20  i = ptr_func(1,2);  // COR ERROR: function pointer must
point on a valid function
21         // COR Warning: return type of function
is INT but a FLOAT was expected
22 }
23
```

In this example, *ptr_func* is a pointer to a function that return a double but it has been initialized to point to a function that returns an int. The understanding of the red error is given in the orange associated COR message.

Wrong Return Type for Arithmetic Functions: COR

This is a check to establish whether that a wrong return type is used for an arithmetic function.

Using arithmetic functions without including <math.h> is compiler dependent in the real world because compiler could associate a integral return type to an implicit function.

However, as arithmetic functions are built-in in PolySpace software, you can face an inconsistency problem <math.h> is not explicitly included in the code file where an arithmetic function is used. All arithmetic function declared in <math.h> are concerned.

Consider the following examples:

Results without <math.h>:

```
1
2 int main(void) {
```

```
3
4 double x;
5 x = cos(2*3.1415); // COR ERROR: return type of
function cos is INT 32 but a float 64 was expected
6 }
```

Results with <math.h>:

```
1 #include <math.h>
2 int main(void) {
3
4 double x;
5 x = cos(2*3.1415);
6 }
```

In the previous example without the definition of <math.h>, cos is declared without prototype and default return type is an int32.

Pointer Within Bounds: IDP

Check to establish whether a reference refers to a valid object (whether a dereference pointer is still within the bounds of the object it intended to point to).

Consider the following example.

```
1
2 #define TAILLE_TAB 1024
3 int tab[TAILLE_TAB];
4 int *p = tab;
5
6 void main(void)
7 {
8
9  int index;
10
11  for (index = 0; index < TAILLE_TAB ; index++, p++)
12  {
13   *p = 0;
14  }
15
```

```
16  *p = 1; // IDP ERROR: reference refers to an invalid object
17 }
```

In the example, the pointer *p* is initialized to point to the first element of the *tab* array at line 4. When the loop is exited, *p* points beyond the last element of the array.

Thus line 16 overwrites memory illegally.

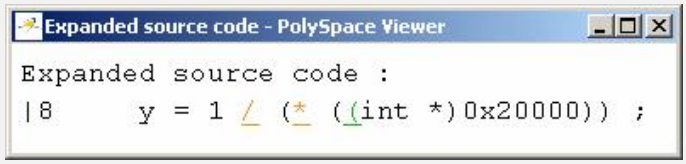
Understanding Addressing

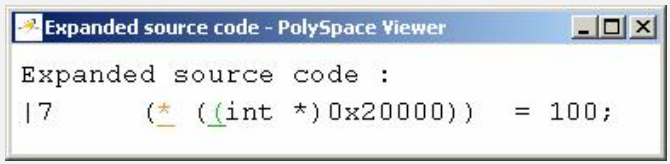
I Systematically Have an Orange Out of Bounds Access On My Hardware Register. Many code verifications exhibit **orange** out of bound checks with respect to accesses to absolute addresses and/or hardware registers.

(Also refer to the discussion on Absolute Addressing)

Here is an example of what such code might look like:

```
#define X (* ((int *)0x20000))
X = 100;
y = 1 / X;  // ZDV check is orange because
     // X ~ [-2^31, 2^31-1] permanently.
     // The pointer out of bounds check is orange because 0x20000
     // may address anything of any length
     // NIV check is orange on X as a consequence
```






```
3 void main (void)
4 {
5 int y;
6
7 X = 100;
8 y = 1 / X;
9
10  }
```

```
int *p = (int *)0x20000;
*p = 100;
y = 1 / *p;  // ZDV check is orange because
     // *p ~ [-2^31, 2^31-1] permanently
     // The pointer out of bounds is orange because 0x20000
     // may address anything of any length
     // NIV check on *p is orange as a consequence
```

This can be addressed by defining registers as regular variables:

Replace	By
`#define X ....`	`int X;`

Replace	By
`int *p;`	`int _p;#define p (&_p)` **Note** Check that the chosen variable name (p in this example) doesn't already exist
`int *p;`	`volatile int _p;int *p = &_p;`

See "Volatile" for a discussion of an approach which will help avoid the orange check on the pointer dereference, but retains the representation of a "full range" variable.

The NULL Pointer Case. Consider the NULL address, viz.

```
#define NULL ((void *)0)
```

- It is illegal to dereference this NULL address;
- 0 **is not** treated as an absolute address.

`*NULL = 100; //` produces a red - Illegal Dereference Pointer (IDP)

Assuming these declarations:-

```
int *p = 0x5;
volatile int y;
```

and these definitions:-

```
#define NULL ((void *) 0)
#define RAM_MAX ((int *)0xffffffff)
```

consider the code snippets below.

```
While (p != (void *)0x1)
 p--; // terminates
```

0x1 is an absolute address, it can be reached and the loop terminates

```
for (p = NULL; p <= RAM_MAX; p++)
{
 *p = 0; // illegal dereference of pointer
}
```

At the first iteration of the loop p is a NULL pointer. Dereferencing a NULL pointer is forbidden.

```
While (p != NULL)
{
 p--;
 *p = 0; // Orange dereference of a pointer
}
```

When p reaches the address 0x0, there is an attempt to considered it as an absolute address

In effect, it is an attempt to dereference a NULL pointer - which is forbidden.

Note In this case, the **check** is **orange** because the execution of the code here is OK (green) until 0x0 is reached (red)

The best way to address this issue depends on the purpose of the function.

- Thanks to the default behavior of PolySpace software, it is easy to automatically stub a function whose purpose is to copy data from/to RAM or to compute a checksum on RAM.
- If a function is supposed to copy calibration data, it should also be stubbed automatically.
- If the purpose of a function is to map EEPROM data to global variables, then a manually written stub is essential to ensure the assignment of the correct initialization values to them.

Comparing Address. PolySpace software only deals with the information referred to by a pointer, and not the physical location of a variable. Consequently it does not compare addresses of variables, and makes no assumption regarding where they are located in memory.

Consider the following two examples of PolySpace verification behavior:

```
int a,b;
if (&a > &b) // condition can be true and/or false
{ } // both branches are reachable
else
{ } // both branches are reachable
```

and

```
int x,z;
void main(void)
{ int i;
 x = 12;
 for (i=1; i<= 0xffffffff; i++)
 {
  *((int *)i) = 0;
 }
 z = 1 / x;  // ZDV green check because PolySpace doesn't consider
     // any relationship between x and its address
}
```

"x" is aliased by no other variable. No pointer points to "x" in this example, so as far as the PolySpace verification is concerned, "x" remains constantly equal to 12.

Understanding Pointers

PolySpace software does not analyze anything which would require the physical address of a variable to be taken into account.

- Consider two variables x and y. PolySpace verification will not make a meaningful comparison of "&x" (address of x) and "&y"

- So, the Boolean (&x < &y) can be true or false as far as PolySpace verification is concerned.

However, PolySpace verification does keep track of the pointers that point to a particular variable.

- So, if ptr points to X, *ptr and X will be synonyms.

Address Alignment: the bitfield Example. Structure size depends on bit alignment.

Consider the following example, where an attempt is made to map a character to a bitfield.

```
struct reg {
 unsigned int a: 5;
 unsigned int b: 3;
};
int main()
{
 volatile unsigned char c;
 struct reg *r;
 r = (struct reg *) &c;
 if (r-> a == 10)
  return 1;
 return 0;
}
```

Consider a 32 bit target architecture (so int are 32 bits, i.e. 4 bytes). The size of a bit field is the size of the type of its elements. In the example above, the elements in the bit field are unsigned int, hence the size is 4 bytes. Since this is greater than 1, the structure reg cannot be contained in the char c.

This can be solved by using the unsigned char type for the elements in the bit field. The size of the bit field is then 1 byte and there is therefore no red error.

```
struct reg {
 unsigned char a: 5;
 unsigned char b: 3;
};
int main()
```

```
{
 volatile unsigned char c;
 struct reg *r;
 r = (struct reg *) &c;
 if (r-> a == 10)
  return 1;
 return 0;
}
```

Note You must also use the option -allow-non-int-bitfield to implement this solution, since this is an extension to the ANSI standard.

How Does malloc Work for PolySpace Verification?. PolySpace verification accurately models malloc, such that both the possible return values of a null pointer and the requested amount of memory are taken into account.

Consider the following example.

```
void main(void)
{
 char *p;
 char *q;
 p = malloc(120);
 q = p;
 *q = 'a'; // results in an orange dereference check
}
```

This code will avoid the orange dereference:

```
void main(void)
{
 char *p;
 char *q;
 p = malloc(120);
 q = p;
 if (p!= NULL)
  *q = 'a'; // results in a green dereference check
}
```

Data Mapping into a Structure . It often happens that structured data are read as a char array. Before manipulating them it might be desirable to map those data into a structure that reflects their organization. In the following example an IDP warning (orange check) at line 22 suggests that the correctness of the code needs to be confirmed.

```
1
2
3 typedef struct
4 {
5  unsigned int MsgId;
6  union {
7   float fltv;
8   unsigned int intv;
9  } Msgbody;
10 } Message;
11
12 int random_int(void);
13 Message *get_msg(void);
14 void wait_idl(void);
15
16 void treatment_msg(char *msg)
17 {
18  Message *ptrMsg;
19
20  ptrMsg = (Message *)msg;
21  if (ptrMsg != NULL) {
22   if (ptrMsg->MsgId) { // IDP Warning: reference may not
refer to a valid object
23    // ...
24   }
25  }
26 }
27
28 int main (void) {
29
30  Message *msg;
31
32  while(random_int()) {
33   msg = get_msg();
```

```
34   if (msg) treatment_msg((char *)msg);
35   wait_idl();
36  }
37  return 0;
38 }
```

Mapping of a small structure into a bigger one. For example, suppose that *p* is a pointer to an object of type *t_struct* and it is initialized to point to an object of type *t_struct_bis*.

Now suppose that the size of *t_struct_bis* is less than the size of *t_struct*. Under these circumstances, it would be illegal to dereference *p* because it would be possible to access memory outside of *t_struct_bis*.

Consider the following example.

```
1 #include <malloc.h>
2
3 typedef struct {
4  int a;
5 union {
6   char c;
7   float f;
8  } b;
9 } t_struct;
10
11 void main(void)
12 {
13  t_struct *p;
14
15  // optimize memory usage
16  p = (t_struct *)malloc(sizeof(int)+sizeof(char));
17
18  p->a = 1; // IDP ERROR: not allowed to deference p
19
20 }
```

Partially allocated pointer (-size-in-bytes). According to the ANSI standard, the whole of a structure must be populated for that structure to be valid. In this case, the pointer is said to be fully allocated. A pointer is said to be partly allocated when only the first part of a structure is populated. In some development environments, that approach is tolerated despite the ANSI stance.

By default, PolySpace verification strictly conforms to the standard and checks for adherence to it. A more tolerant approach can be specified by using the -size-in-bytes option. So, depending on the -size-in-bytes option, when a partially allocated pointer is encountered during a PolySpace verification, the first elements of the allocated object may or may not be considered as valid.

First consider the following example. (A second example follows it to illustrate how this might apply to pointer arithmetic within a structure)

```
1  typedef struct _little { int a; int b; } LITTLE;
2  typedef struct _big { int a; int b; int c; } BIG;
3
4 int main(void)
5  {
6   BIG *p = malloc(sizeof(LITTLE));
7   volatile int y;
```

With -size-in-bytes option

```
9   if (p==((void *)0)) return 0;
10   if(y) { p->a = 0; } // green
11   if(y) { p->b = 0; } // green
12   if(y) { p->c = 0; } // red
}
```

Default launching option

```
9   if(y) { p->a = 0 ; } // red
10   if(y) { p->b = 0 ; } // red
11   if(y) { p->c = 0 ; } // red
12
13   if (p==((void *)0))
14     return 0;
15   else
```

```
16     return 1; // dead code
17   return 1;
18  }
```

With the standard launching option, a pointer that has not been allocated to a complete structure is considered invalid, or NULL (as shown in the dead code).

Pointer to a structure field. According to the ANSI C standard, pointer arithmetic is to be independent of the size of the object (structure or array) to which the pointer points. By default, PolySpace verification strictly conforms to the standard and checks for adherence to it.

In some development environments an approach that does not recognize that requirement is tolerated, despite the ANSI stance. Under those circumstances, results are likely to include **red** pointer out of bounds **checks** unexpectedly.

A more tolerant approach can be specified at launch time. Consider the following examples.

```
char *p; // the size of the object pointed to is unknown,
// but arithmetic on this pointer is well defined.
// p = p + 5; will increment the location pointed to by
5 bytes (if the
size of a char is 1 byte)
int x;  // assuming that an int is 4 bytes
p = &x; *p = 0; // the first byte of x
p++; *p = 0;  // the second byte of x
p++; *p = 0;  // the third byte of x
p++; *p = 0;  // the fourth byte of x
p++; *p = 0;  // an out of bound access
```

For structures, the same behavior can be applied.

```
struct { int a; int b; } x;
char *p = &x.a; // the pointed object is not the structure
but the field
*p = 0; // it is the first byte of x.a
p++; *p = 0;  // it is the second byte of x.a
p++; *p = 0;  // it is the third byte of x.a
```

```
p++; *p = 0;  // it is the fourth byte of x.a
p++; *p = 0;  // here is an out of bound access because
we are out of the field
```

If you wish to tolerate an approach which allows a pointer to go from one field to another, you can do so by using the -size-in-bytes option **together with** the -allow-ptr-arith-on-struct option . When a pointer points to a field in a structure, you will then be allowed to access other fields from this pointer. Note that as a consequence, any other "out of bound" accesses in the code will be ignored.

An alternative solution is to make your variable point to the structure rather than to the field, as follows:

```
struct { int a; int b; } x;
char *p = &x;  // the pointed object is the structure
*p = 0;    // we are modifying x.a (first byte)
p++; *p = 0;  // we are modifying x.a (second byte)
p++; *p = 0;   // we are modifying x.a (third byte)
p++; *p = 0;  // we are modifying x.a (fourth byte)
p++; *p = 0;  // we are modifying x.b (fifth byte)
```

A further alternative is to follow the ANSI C recommendation to use the “offsetof()” function, which jumps to the corresponding offset within the structure:-

```
#include <stddef.h>
typedef struct _m { int a; int b; } S;
S x;
char *p = (char *) &x + offsetof(S,b); // points to field b
```

I have a red when reading a field of one structure. Consider the following example.

```
5 typedef struct {
6  unsigned char c1;
7  unsigned char c2;
8 } my_struct;
9
10 int main(void)
11 {
```

```
12  my_struct v;
13  unsigned short x=0,y=0;
14
15  v.c1=9;
16  v.c2=15;
17  x = *((unsigned short *)&v.c1);
```

Just like the example in “Pointer to a structure field” on page 2-33, the object pointed to is the field in the structure, not the structure itself. Therefore, it is only possible to navigate inside this field. A short variable occupies more memory than a char, so it is a red pointer out of bounds.

This can be addressed by replacing

```
x = * ((unsigned short *) &v.c1);
```

with

```
y = (v.c1 << sizeof(v.c2)*8 ) | v.c2;
```

This solution also ensures that the code is no longer target dependent.

Non Termination of Call or Loop

NTC and NTL are informative red (or orange) checks.

- They are the only red checks which can be filtered out as shown below
- They don’t stop the verification
- As for other red checks, code found after them are gray (unreachable)
- These checks may only be red. There are no “orange” NTL or NTC checks.
- They can reveal a bug, or can simply just be informative

NTL	In a Non Terminating Loop, the break condition is never met. Here are some examples.
	
	while(1) { function_call(); } // informative NTL
	
	while(x>=0) {x++; } // where x is an unsigned int. This may reveal a bug?
	
	for(i=0; i<=10; i++) my_array[i] = 10; // where "int my_array[10];" applies. This red NTL reveals a bug in the array access, flagged in orange
	
	ptr = NULL; for(i=0; i<=100; i++) *ptr=0; // the first iteration of the loop is red, and therefore it is flagged as an NTL. The "i++" will be gray, because the first iteration crashed.

NTC	Suppose that a function calls f(), and that function call is flagged with a red NTC check. There could be five distinct explanations: • "f" contains a red error; • "f" contains an NTL ; • "f" contains an NTC; • "f" contains an orange which is context dependant; that is, it is either red or green. For this particular call, it makes the function "f" crash. • "f" is a mathematical function, such as sqrt, acos which has always an invalid input parameter Remember, additional information can be found when clicking on the NTC

Note A sqrt check is only colored if the input parameter is **never** valid. For instance, if the variable x may take any value between -5 and 5, then sqrt(x) has no color.

The list of constraints which cannot be satisfied (found by clicking on the NTC check) represents the variables that cause the red error inside the function.

The (potentially) long list of variables can help to understand the cause of the red NTC, as it shows each condition causing the NTC

- where the variable has a given value; and
- where the variable is not initialized. (Perhaps the variable is initialized outside the set of files under verification?).

If a function is identified which is not expected to terminate (such as a loop or an exit procedure) then the -known-NTC function is an option. You will find all the NTCs and their consequences in the k-NTC facility in the Viewer, allowing you to filter them.

Non Termination of a Call: NTC

This is a check to establish whether a procedure call returns. It is not the case when the procedure contains an endless loop or a certain error, or if the procedure calls another procedure which does not terminate. In the latter instance, the status of this check is propagated to caller.

```
1
2
3 void foo(int x)
4 {
5  int y;
6  y = 1 / x; // Warning ZDV: its depends of the context
7  while(1) { // NTL ERROR: loop never terminates
8   if ( y != x) {
9    y = 1 / (y-x);
10   }
11  }
12 }
13
14 void main(void) {
15  volatile int _x;
16
17  if (_x)
18   foo(0); // NTC ERROR: Zero DiVision (ZDV) in foo
19  if (_x)
20   foo(2); // NTC ERROR: Non Termination Loop (NTL) in foo
21
```

```
22 }
23
```

In this example, the function *foo* is called twice in *main* and neither of these 2 calls ever terminates.

1. The first never returns because a division by zero occurs at line 6 (bad argument value),
2. The second never terminates because of an infinite loop (red NTL) at line 7.

Also with reference to the example and as an aside, note that by using either the -context-sensitivity "foo" option or the -context-sensitivity-auto option at launch time it would be possible for PolySpace verification to show explicitly that a ZDV error comes from the **first** call of *foo* in *main*.

Note An NTC check can only be red or uncolored, unless you use the -context-sensitivity option. If you use the -context-sensitivity option, NTC checks can also be orange.

Known Non-Termination of a Call: k-NTC

By using the -known-NTC option with a specified function at launch time it is possible to transform an NTC check to a k-NTC check. Like NTC checks, k-NTC checks are propagated to their callers. Functions designed not to terminate can then be filtered out through the use of the appropriate filter in the viewer.

Consider the following example, supposing that **-know-NTC "SysHalt"** option has been applied at launch time.

```
1
2 /* external get data function */
3 extern int get_data(int *ptr,void *data);
4 extern int printf (const char *, ...);
5
6 // known NTC function
7 void SysHalt(int value)
8 {
```

```
 9  printf("Halt value %d",value);
10   while (1) ; // NTL ERROR: Loop Never Terminate
11 }
12
13 #define OK 1
14 int main(void)
15 {
16  int data, *ptr = NULL;
17  int status = OK;
18
19  // get next store
20  status = get_data(ptr,(void *)&data);
21  if (status != OK)
22   SysHalt(status); // k-NTC check: Call never
terminate
23
24  return(0);
25 }
```

In the example, the relevant NTC check is converted to a k-NTC one.

Non Termination of Loop: NTL

This is a check to establish whether a loop (for, do-while or while) terminates. Consider the following example:

```
1
2 // Function prototypes
3 void send_data(double data);
4 void update_alpha(double *a);
5
6 void main(void)
7 {
8  volatile double _acq;
9  double acq, filtered_acq, alpha;
10
11  // Init
12  filtered_acq = 0.0;
13  alpha = 0.85;
14
15  while (1) { //NTL ERROR: Non Termination Loop
```

```
16   // Acquisition
17   acq = _acq;
18   // Treatment
19   filtered_acq = acq + (1.0 - alpha) * filtered_acq;
20   // Action
21   send_data(filtered_acq);
22   update_alpha(&alpha);
23  }
24 }
```

In the example, the continuation condition is always true and the loop will never exit. PolySpace verification will raise an error in trivial examples such as this, and in much more complex circumstances.

Consider this second verification. When an error is found inside a for, do-while, or while loop, PolySpace will not continue to propagate it.

```
1
2 void main(void)
3 {
4  int i;
5  double twentyFloat[20];
6
7  for (i = 0; i <= 20; i++) {
  // NTL ERROR: propagation of OBAI ERROR
8   twentyFloat[i] = 0.0; // OBAI Warning: 20 verification with i
in [0,19] and one ERROR with i = 20
9  }
10 }
```

At line 8 in this second example, the **red** OBAI related to the **21st** execution of the loop has yielded the **orange check**. The 20 first executions would be no problem, so this **orange** warning represents a combination of **red** and **green** checks.

Note An NTL check can only be **red** or uncolored, unless you use the -context-sensitivity option. If you use the -context-sensitivity option, NTL checks can also be orange.

Tooltips for NTL Checks. Tooltips provide range information in the viewer, including the number of iterations for loops.

There are 2 possible situations:

- **Loops that terminate** – A tooltip gives the number of iterations of the loop. For example, `for (i=0; i<10; i++)`, a tooltip on the `for` keyword says `Number of iteration(s): 10`.
- **Non–terminating loops** — The NTL check contains information about the maximum number of iterations that can be done. This number is an overset of the real number of iterations (which may be lower).

  For example:

  - **Failure at a given iteration**, `for (i=0; i<10; i++) y = 2 / (i - 5);` — The NTL check on the `for` keyword says: `Number of iteration(s): 6`

    This means that the loop fails at the 6th iteration, which can help you find the orange check that contains the failure.

  - **Infinite loop** `x = 0; while (x >= 0) y = 2;` — The NTL check on the `for` keyword says: `Number of iteration(s): 0..?`

    This means that the loop has an unknown number of iterations (up to an infinite number). It does not mean that the loop *is* an infinite loop, but that it *may* be an infinite loop. You would also get `0..?` on the loop `while (1) { if (random) break; }`.

Arithmetic Expressions: NTC

This is a check to establish whether standard arithmetic functions are used with valid arguments, as defined in the following:

- Argument of *sqrt* must be positive (ISO®/IEC 9899 section 7.5.5.2)
- Argument of *tan* must be different from *pi/2* modulo *pi* (ISO/IEC 9899 section 7.5.2.7)
- Argument of *log* must be strictly positive (ISO/IEC 9899 section 7.5.4.4)
- Argument of a*cos* and a*sin* must be within [-1..1] (ISO/IEC 9899 sections 7.5.2.1 and 7.5.2.2)

- Argument of *exp* must be less than or equal to 709 (ISO/IEC 9899 section 7.5.4.1)
- Argument of *atanh* must be within]-1..1[(ISO/IEC 9899 section 7.12.5.3)
- Argument of *acosh* must be greater or equal to 1 (ISO/IEC 9899 section 7.12.5.1)

A domain error (such that *errno* returns *EDOM*) occurs if an input argument is outside the domain over which the mathematical function is defined. A range error occurs (such that *errno* returns *ERANGE*) if the result cannot be represented as a double value. In the latter case, the function returns *0* if the result is too small, or *HUGE_VAL* with the appropriate sign if it is too big.

Consider the following example

```
1
2 #include <math.h>
3 #include <assert.h>
4
5 extern int random_int(void);
6
7 int main(void)
8 {
9
10  volatile double dbl_random;
11  const double dbl_one = 1.0;
12  const double dbl_mone = -1.0;
13
14  double sp      = dbl_random;
15  double p      = dbl_random;
16  double sn      = dbl_random;
17  double n      = dbl_random;
18  double no_trig_val_neg = dbl_random;
19  double no_trig_val_pos = dbl_random;
20  double pun      = dbl_random;
21  double res;
22
23  // assert is used here to redefine range values of variables
24  assert(sp > 0.0);
25  assert(p >= 0.0);
```

```
26  assert(sn < 0.0);
27  assert(n <= 0.0);
28  assert(pun < 1.0);
29  assert(no_trig_val_neg < -1.0); assert(no_trig_val_pos > 1.0);
30
31  if (random_int()) res = sqrt(sn);       // NTC ERROR:
need argument positive
32  if (random_int()) res = asin(no_trig_val_neg);  // NTC ERROR:
need argument in range [-1..1]
33  if (random_int()) res = asin(no_trig_val_pos);  // NTC ERROR:
need argument in range [-1..1]
34  if (random_int()) res = acos(no_trig_val_pos);  // NTC ERROR:
need argument in range [-1..1]
35  if (random_int()) res = acos(no_trig_val_neg);  // NTC ERROR:
need argument in range [-1..1]
36  if (random_int()) res = tan(1.5707963267948966); // NTC ERROR:
need argument in range ]-pi/2..pi/2[
37  if (random_int()) res = log(n);       // NTC ERROR:
need argument strictly positive
38  if (random_int()) res = exp(710);       // NTC ERROR:
need argument less or equal to 709
39
40  // No information about asin or acos because of random value
41  if (random_int()) {
42   res = asin(dbl_random);
43   res = acos(dbl_random);
44  }
45
46  // hyperbolic functions are available in the float range
47  if (random_int()) {
48   res = cosh(710);
49   res = cosh(10.0);
50   assert (res < 1.0);
51  }
52  if (random_int()) res = sinh(710);
53  if (random_int()) {
54   res = tanh(1.0);
55   assert (res > -1.0 && res < 1.0);
56  }
57
```

```
58  // inverted hyperbolic functions
59  if (random_int()) res = acosh(pun);      // NTC ERROR:
Need argument >= 1
60  else res = acosh(1.0);
61  if (random_int()) res = atanh(no_trig_val_neg);  // NTC ERROR:
Need argument in ]-1..1[
62  if (random_int()) res = atanh(no_trig_val_pos);  // NTC ERROR:
Need argument in ]-1..1[
63  if (random_int()) res = atanh(dbl_mone);    // NTC ERROR:
Need argument in ]-1..1[
64  if (random_int()) res = atanh(dbl_one);    // NTC ERROR:
Need argument in ]-1..1[
65
66  return 0;
67 }
68
```

sqrt, *tan*, *asin*, *acos*, *exp* and *log* errors are derived directly from the mathematical definition of functions. PolySpace verification highlights any definite problems by means of an NTC to show that this is where execution would terminate. No NTC information is delivered when PolySpace cannot determine the exact value of the argument, (for *asin* and *acos* at lines 42 and 43). No range restriction is currently made for hyperbolic functions.

Caution Due to a lack of precision in some areas, PolySpace verification is not always able to indicate a red NTC check on mathematical functions even where a problem exists. In the following example involving a *sqrt* function, neither an orange nor a red check is shown on line16 even though the variable *val2* is negative.

By default it is important to consider each call to any mathematical functions as though it had been highlighted by an **orange check**, and could therefore lead to a runtime error.

```
1
2 #include <math.h>
3
4 extern int random_int(void);
```

```
5
6 int main(void)
7 {
8
9  double val1, val2;
10
11  int i;
12  val2 = 5.0;
13  for (i = 0 ; i < 10 ; i++) {
14   val2 = val2 - 1.0;
15  }
16  val1 = sqrt(val2); // No check on sqrt
17  return ((int)val1);
18 }
19
```

Unreachable Code: UNR

This is a check to establish whether different code snippets (assignments, returns, conditional branches and function calls) are dead, such that they can never be accessed during the normal execution of the software. Dead, or Unreachable, code is represented by means of a gray coding on every check, with supplementary UNR checks also being added.

Consider the following example.

```
1
2 #define True 1
3 #define False 0
4
5 typedef enum {
6  Intermediate, End, Wait, Init
7 } enumState;
8
9 // pure stub
10 int intermediate_state(int);
11 int random_int(void);
12
13 int State (enumState stateval)
14 {
15  volatile int random;
```

```
16  int i;
17  if (stateval == Init) return False;
18  return True;
19 }
20
21 int main (void)
22 {
23  int i, res_end;
24  enumState inter;
25
26  res_end = State(Init);
27  if (res_end == False) {
28   res_end = State(End);
29   inter = (enumState)intermediate_state(0);
30   if (res_end || inter == Wait) {    // UNR code on inter
== Wait
31    inter = End;
32   }
33   // use of I not initialized
34   if (random_int()) {
35    inter = (enumState)intermediate_state(i); // NIV ERROR
36    if (inter == Intermediate) {    // UNR code because
of NIV ERROR
37     inter = End;
38    }
39   }
40  } else {
41   i = 1; // UNR code
42   inter = (enumState)intermediate_state(i);  // UNR code
43  }
44  return res_end;
45 }
46
```

The example illustrates three possible reasons why code might be unreachable, and hence be colored **gray**:

- At line 30 the first part of a two part test is always true. The other part is never evaluated, following the standard definition of logical operator "||".

- The piece of code after a red error is never evaluated by PolySpace software. The call to the function on line 35 and the line following it are considered to be dead code. Correcting the red error and relaunching would allow the color to be revised.
- At line 27, the test is always true (*if-{* part), and the first branch is always executed. Consequently there is dead code in the other branch (i.e. in the *else* part at lines 41 to 42).

Inspection Points

You can create inspection points in the code that provide range information.

For example:

```
#pragma Inspection_Point <var1> <var2>
```

where *var1* and *var2* are scalar variables, instructs the PolySpace verification to provide range information for `var1` and `var2` at that point in the code. You see this information in a tooltip message when you place your cursor over `var1` or `var2`.

3

Approximations Used During Verification

Why PolySpace Verification Uses Approximations

In this section...

What is Static Verification

PolySpace software uses *static verification* to prove the absence of runtime errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as runtime debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the PolySpace verification are true for all executions of the software.

PolySpace verification works by approximating the software under verification, using safe and representative approximations of software operations and data.

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{    tab[i] = foo(i);
}
```

To check that the variable 'i' never overflows the range of 'tab' a traditional approach would be to enumerate each possible value of 'i'. One thousand checks would be needed.

Using the static verification approach, the variable 'i' is modelled by its variation domain. For instance the model of 'i' is that it belongs to the [0..999] static interval. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborated models are also used for this purpose).

Any approximation leads by definition to information loss. For instance, the information that 'i' is incremented by one every cycle in the loop is lost. However the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the variation domain of 'i' is smaller than the range of 'tab'. Only one check is required

to establish that – and hence the gain in efficiency compared to traditional approaches.

Static code verification does have an exact solution, but that solution is generally not practical, as it would generally require the enumeration of all possible test cases. As a result, approximation is required.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that PolySpace verification works by performing upper approximations. In other words, the computed variation domain of any program variable is always a superset of its actual variation domain. The direct consequence is that no runtime error (RTE) item to be checked can be missed by PolySpace verification.

Approximations Made by PolySpace Verification

In this section...

Volatile Variables

Volatile variables are potentially uninitialized and their content is always full range.

```
2 int volatile_test (void)
3 {
4  volatile int tmp;
5  return(tmp); // NIV orange: the variable content is full range
[-2^31;2^31-1]
6 }
```

In the case of a global variable the content would also be full range, but the NIV check would be green.

Structures with Volatile Fields

In this example, although only the b field is declared as volatile, in practice any read access to the "a" field will be full range and orange.

```
2 typedef struct {
3  int a;
4  volatile int b;
5 } Vol_Struct;
```

Absolute Addresses

Both reading from, and writing to, an absolute address leads to warning checks on the pointer dereference. An absolute address is considered as a volatile variable.

```
Val = *((char *) 0x0F00); // NIV and IDP orange: access to an
absolute address
```

Pointer Comparison

PolySpace verification is a static tool verifying source code. Memory management concerns dynamic considerations, and the characteristics of particular compilers and targets. PolySpace verification therefore doesn't consider where objects are actually implanted in memory

```
5  int *i, *j, k;
6  i = (int *) 0x0F00;
7  j = (int *) 0x0FF0;
8
9  if ( i < j) // the condition can be true or false
10   k = 12; // this line is reachable
11  else
12   k = 23; // this line is reachable too.
```

Its the same situation if "i" and "j" points to real variable

```
6  i = & one_variable;
7  j = & another_one;
9  if ( i < j) // the condition can still be true or false
```

Shared Variables

At the minimum, a shared variable contains a union of all ranges it can contain among the application. At the maximum, the variable will be full range.

```
12 void p_task1(void)
13 {
14  begin_cs();
15  X = 0;
16  if (X) {
17   Y = X;      // Verified NIV, although it should be gray
```

```
18   assert (Y == 12);  // Warning assert, although it should be gray
19  }
20  end_cs();
21 }
22
23 void p_task2(void)
24 {
25  begin_cs();
26  X = 12;
27  Y = X + 1;      // PolySpace considers [Y==1] or [Y==13]
28  if (Y == 13)
29   Y = 14;
30  else
31   Y = X - 1 ;    // this line should be gray
32  end_cs();
33 }
```

Trigonometric Functions

With trigonometric functions, such as sines and cosines, verification sometimes assumes that the return value is bound between the limits of that function, regardless of the parameter passed to it. Consider the following example, which uses acos, sin and asin functions.

```
7  double res;
8
9  res = sin(3.141592654);
10 assert(res == 0.0); // Range is [-1..1]
11
12 res = acos(0.0);
13 assert(res == 0.0); // Range always in [0..pi]
14
15 res = asin(0.0);
16 assert(res == 0.0); // Always gives [0.0]
```

Unions

In some situations, unions can help you construct efficient code. However, unions can cause issues for code verification, for example:

- **Padding** – Padding might be inserted at the end of an union.

- **Alignment** – Members of structures within a union might have different alignments.
- **Endianness** – Whether the most significant byte of a word could be stored at the lowest or highest memory address.
- **Bit-order** – Bits within bytes could have both different numbering and allocation to bit fields.

These issues can cause PolySpace verification to lose precision when structure unions are considered. In fact, these kinds of implementation are compiler dependant. Conversions from one type a union to another will cause a loss of precision on two checks:

- Is the other field initialized? Orange NIV
- What is the content of the other field? Full range

```
typedef union _u {
int a;
char b[4]; } my_union;
my_union X;

X.b[0] = 1; X.b[1] = 1; X.b[2] = 1; X.b[1] = 1;
if (X.A == 0x1111)
else // both branches are reachable
```

Constant Pointer

To increase PolySpace precision where pointers are analyzed, replace

```
const int *p = &y;
```

with:

```
#define p (&y)
```

4

Examples

Complete Examples

In this section...

Simple C Example

```
polyspace-c \
 -prog myCproject \
 -O1 \
 -I /home/user/includes \
 -D SUN4 -D USE_FILES \
```

Apache Example

Here is a script for verifying the code for Apache (after proper formatting). The source code is in C and the compilation is for a Sun™.

Note The use of O0 to reduce verification time.

```
polyspace-c \ \
 -target sparc \
 -prog Apache \
 -keep-all-files \
 -allow-undef-variables \
 -continue-with-red-error \
 -OO \
 -D PST \
 -D __GNUC_MINOR__=6 -D SOLARIS2=270 -D USE_EXPAT \
 -D NO_DL_NEEDED \
```

```
 -I sources \
 -I /usr/local/pst/include.sparc \
 -I /usr/include \
 -results-dir RESULTS
```

cxref Example

Here is another C launch command. The compilation is for Linux. Note the escape characters, allowing quoted strings to be used as compiler defines.

```
polyspace-c \
 -OS-target linux \
 -prog cxref \
 -OO \
 -I `pwd` \
 -I sources \
 -I <<PolySpace_Verifer_InstallPath>>/include/include.linux \
 -D CXREF_CPP='\"/usr/local/gcc/bin/cpp\"' \
 -D PAGE='\"A4\"' \
 -results-dir RESULTS
```

T31 Example

Another simple C launcher. There are a couple of tasks and compilation is for an m68k.

```
polyspace-c \
 -target m68k \
 -entry-points task_callback_main,task_tcp_main,cdtask_depm_main,
task_receiver \
 -to pass1 \
 -prog T31 \
 -OO \
 -results-dir `pwd`/RESULTS_31 \
 -keep-all-files
```

Dishwasher1 Example

Another C example. This one is for the c-167 and has tasks protected by critical section.

```
polyspace-c \
 -target c-167 \
```

```
 -entry-points periodic,pst_main \
 -D PST -D const= -D water= \
 -from scratch \
 -to pass4 \
 -critical-section-begin "critical_enter:cs1" \
 -critical-section-end "critical_exit:cs1" \
 -prog dishwasher1 \
 -I `pwd`/sources \
 -OO \
 -keep-all-files \
 -results-dir RESULTS
```

Satellite Example

A C example with tasks and critical sections.

```
polyspace-c
 -target c-167 \
 -entry-points ctask0,ctask1,ctask2,ctask3,interrupts \
 -O2 \
 -keep-all-files \
 -from scratch \
 -critical-section-begin "DisableInterrupts:sc1" \
 -critical-section-end "EnableInterrupts:sc1" \
 -ignore-constant-overflows \
 -include `pwd`/sources/options.h \
 -to pass4 \
 -prog satellite \
 -I `pwd`/sources \
 -results-dir RESULTS
```