Evaluation guide for C-RUN

Version 20170428

Contents

What’s in this guide?

This guide is intended as a help in evaluating the runtime analysis product C-RUN that is available as an add-on to IAR Embedded Workbench for ARM and for Renesas RX.

The guide contains general guidelines on how to set up C-RUN in an evaluation context and things to consider when going forward with a more thorough evaluation.

What you need:

To follow this guide you need for one of the following:

- IAR Embedded Workbench for ARM, version 7.30 or later (Standard, Cortex-M or time-limited evaluation edition)
- IAR Embedded Workbench for RX, version 3.10 of later (Standard or time-limited evaluation edition)

Note: *KickStart (size-limited evaluation) and Baseline editions of IAR Embedded Workbench do not work with C-RUN.*

C-RUN licensing considerations

You can evaluate C-RUN in code size limited mode without taking any special action. This mode is mainly for evaluating the C-RUN technology on smaller pieces of code and to explore the integration in to IAR Embedded Workbench. The size-limited mode is not intended for use in a production environment; please get in contact with your IAR Systems representative for access to a non-limited evaluation license.

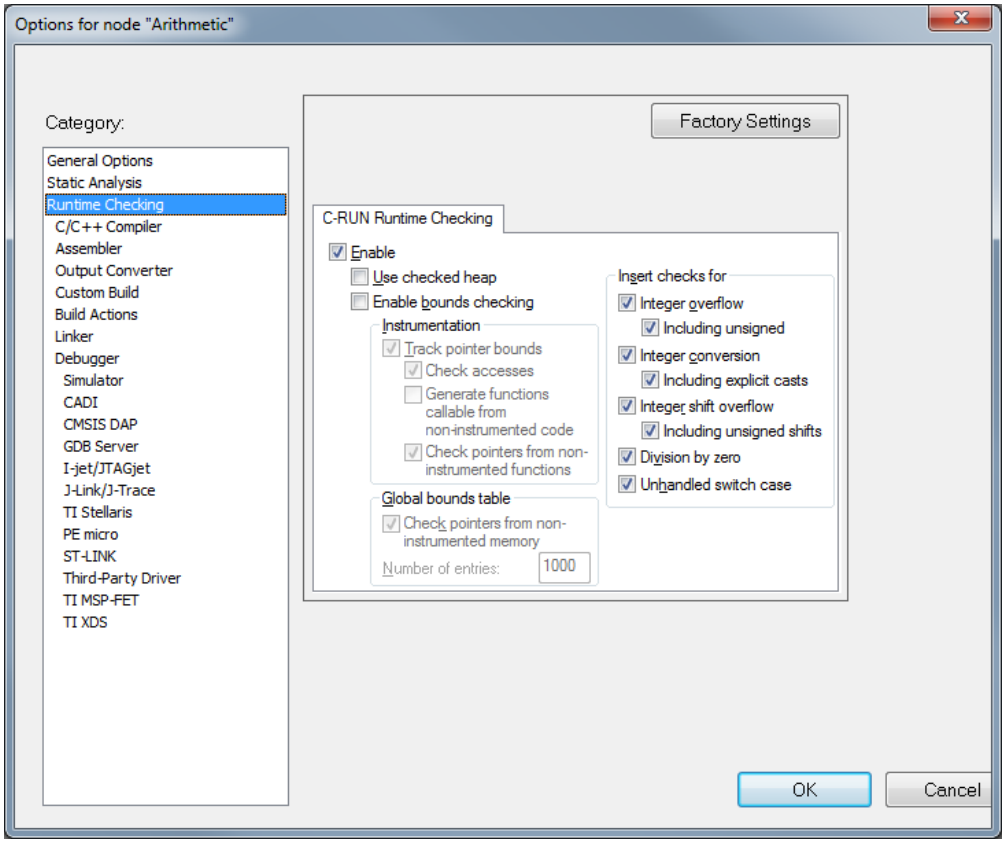
How to proceed?

A number of example projects that demonstrate the various features of the C-RUN runtime checking technology are available together with this guide. Each example can be treated as a lab exercise by following the step-by-step instructions, or they can be used as reference and inspiration for experimentation.

Arithmetic checking

The first example takes a look at the most straightforward functionality in C-RUN, the arithmetic checks. These checks include integer overflow and conversion errors, shift errors and unhandled switch cases. To run the example, follow the steps below.

1. Start IAR Embedded Workbench, open the workspace *C-RUN.eww* and choose the *Arithmetic project*.
2. Read the comment at the top of the example program in *Arithmetic.c*
3. Make sure that all the C-RUN arithmetic checks are enabled, see the screenshot below. Note how checks for unsigned overflow, explicit casts and unsigned shift overflow can be turned off.

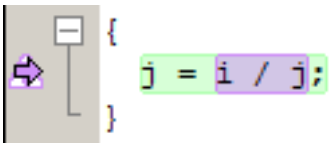


4. Build and run the program in C-SPY.

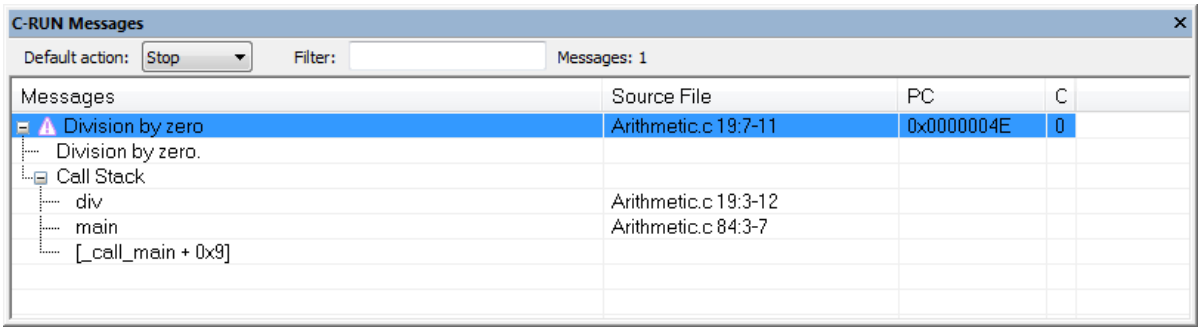
(continued on next page)

Arithmetic checking

5. The execution will stop, highlighting the following line of code:



The highlighted statement has triggered a runtime error and the exact part of the statement that triggered the error is further highlighted. A new window will also open, displaying the cause of the error and a call stack trace.



The error and the call stack are clickable for navigation to the corresponding source locations.

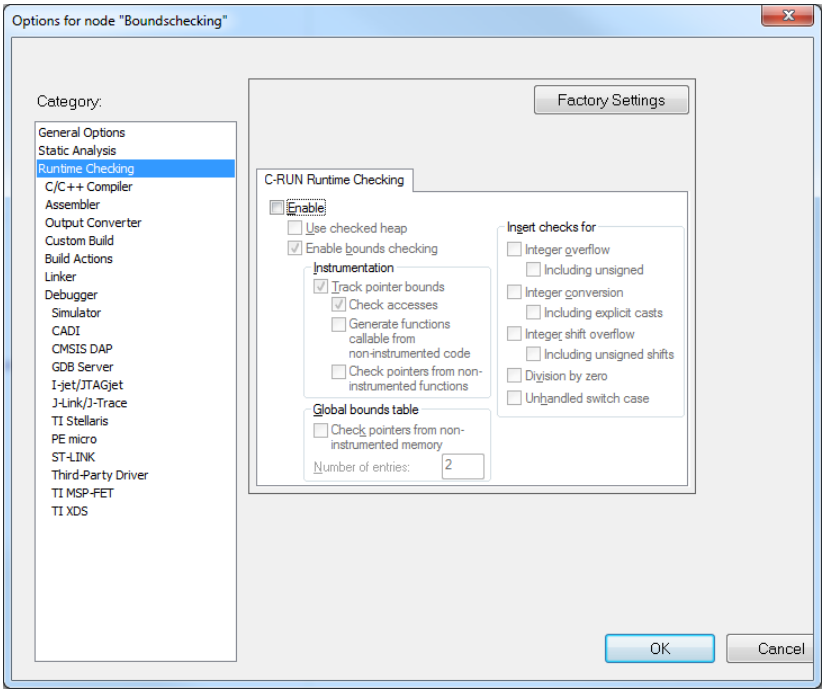
6. Continue execution and examine each reported error.
7. Disable the options for unsigned overflow, explicit casts and unsigned shift overflow and build and re-run the program. Notice that the number of reported issues is lower.
   It is in many situations convenient and efficient to rely on e.g. the overflow behavior for unsigned integers, so the checks can easily be turned off.
8. Default is for execution to stop for each error. This can be changed by selecting another choice in the *Default action* field. You can for example try the *Log* alternative.
9. Messages can be filtered, in case there are certain messages or message types that you do not want to see. See the section *Creating rules for messages* in the *C-SPY Debugging Guide*. Right-clicking on an error message in the C-RUN messages window brings up a menu with choices for filtering.

Note: C-RUN windows can be opened via menu **View>C-RUN**.

Bounds checking

The second example takes a look at the bounds checking. To run the example, follow the steps below.

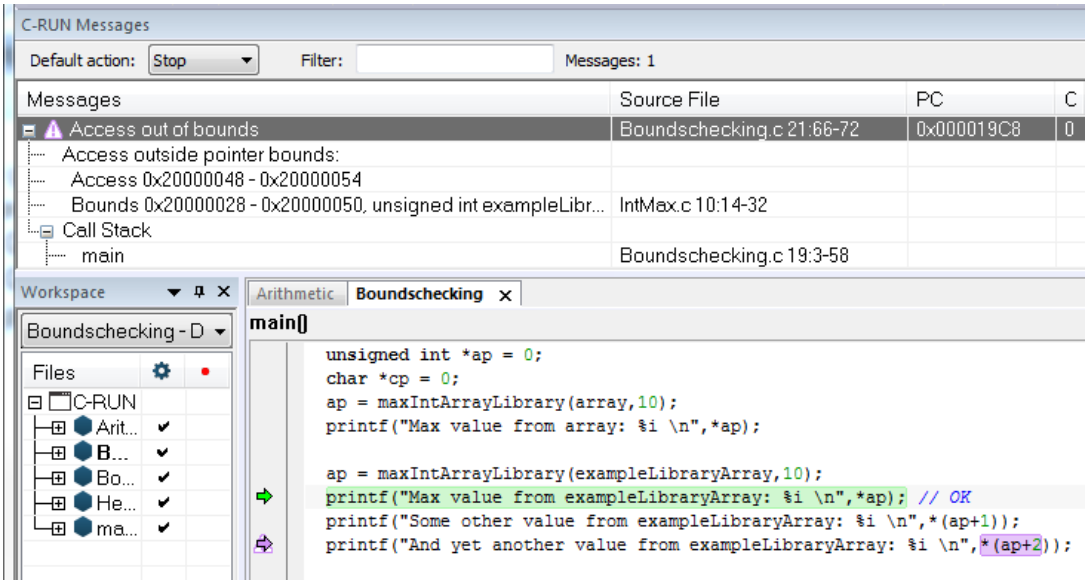
1. Choose the *Boundschecking* project.



2. Check that bounds checking is not enabled, either by deselecting the *Enable* check box, or the *Enable bounds checking* check box. Build and run the program in C-SPY. Note that there are no reported error of any kind from the execution although if inspected it seems that some of the pointer accesses might be erroneous.

3. Enable the bounds checking option and build and run again. If *Stop* is the default action for C-RUN messages, the execution will stop and show the following information (go to next page):

Bounds checking



4. Notice how the out-of-bounds access **(ap+2)* is highlighted, but the execution is stopped already at the first *printf()* statement. This is because the compiler can determine that the pointer values used in all the *printf()* statements are related and can be tested against the bounds in one go.

5. Let the execution continue. It will stop at the last statement of the program, indicating that the assignment is done out-of-bounds. From a bounds checking perspective dynamically allocated memory is no different from local pointers, static buffers or buffers on the stack.

6. In the option dialog for runtime checking there is a field for *Number of Entries*. Pointers that can be accessed trough other pointers need to have information stored in a table in memory. This field lets you decide the number of slots in the table. If you leave the field blank you do not give a size and the table will be huge (4k slots!). The number of slots you need is often fairly low, so you can experiment with lowering the number. In case the number is too low you will get an error message in runtime when there are no available slots.

7. Set the *Number of entries* field to one and build and run the program.

8. In this example there is only one pointer that needs to be kept in the table, so one slot is enough.

Bounds checking and libraries

As we saw in the previous Bounds checking example, when all pointer-manipulating code in a project is written in C/C++ and all code is compiled within the project, bounds checking is very straightforward – just enable it and build and run. However, if some of the code that manipulates pointers is for example residing in a pre-built library or written in assembly language and the library/assembly code accepts or passes pointers to your code the situation is a little bit more complex.

In this example we will look at how to deal with such a situation. The example program is very similar to the one used in example 2, but we have broken out the *IntMax.c* file and put in its own library project to illustrate how to deal with pointers and pointer arguments that pass through interfaces between code that is bounds checked and code that is not.

1. Choose the *Boundschecking2* project. This project is very similar to the previous bound checking project but this project has a prebuilt maxlib library included instead of the *IntMax.c* file. The file *Boundschecking_2.c* is very similar to *Boundschecking.c* in the previous example, but there are a lot more comments and the code doing a malloc and out-of-bounds access at the end is not included.

2. As part of the *main* project, there is a file *readme.txt* that explains the setup in the project and how bounds checking can be utilized also for code over which you have no control.

   Before proceeding to step 3, it is highly advised to read this file. But skip the part about *ReportCheckFailedStdout.c*, at least for now; we will get back to it in the next example.

   (continued on next page)

Bounds checking and libraries

3. Make sure that the *DoNotCheckPointersFromNonInstrumentedCode* build configuration is selected for the *main* project and build and run the code in C-SPY. You should get no reported C-RUN errors and no other indications that something is wrong.

4. Close C-SPY and take a look at the options page for runtime checking. Then change build configuration to *CheckPointersFromNonInstrumentedCode* and build and run the code again.

5. This time you should se one C-RUN message for the third *printf()* statement. Compare the use of *__as_make_bounds()* for the pointer *ap* to how the bounds are set for the pointer used in the next *printf()* statement. In this configuration the options for checking pointer from non-checked code and memory are set. We have also defined a project specific symbol to control the use of *__as_make_bounds().*

6. Comment out one of the calls to *__as_make_bounds()* and build and run again. Did it change the number of C-RUN messages?

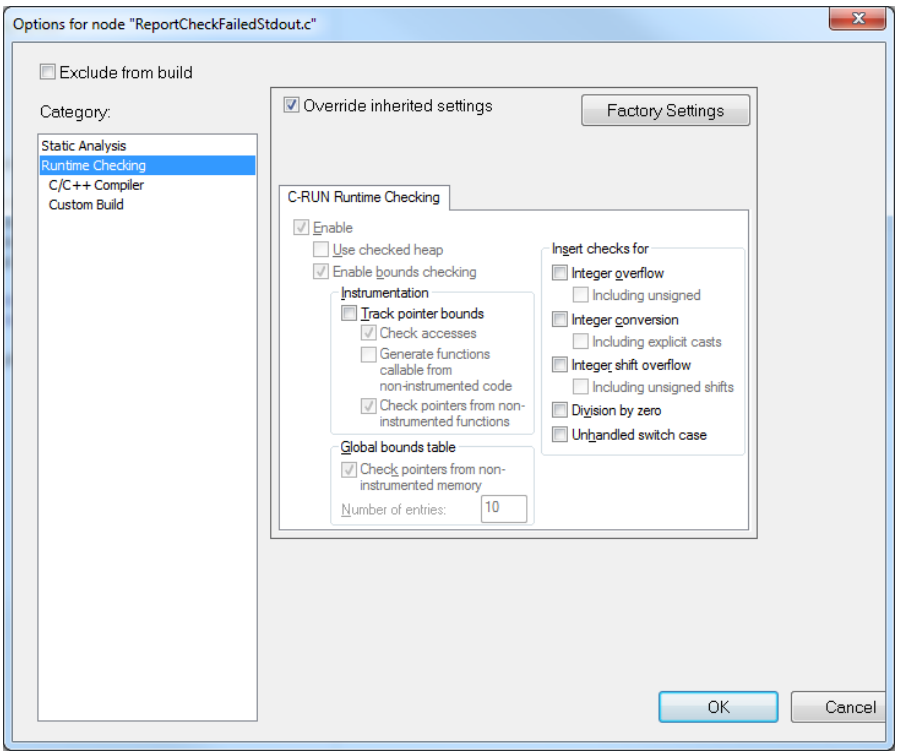
Read the section *Compiler and linker reference for C-RUN* in the *C-SPY debugging guide* for more information.

Using your own reporting

Sometimes it is not possible to execute an application in C-SPY. This might be due to the need for electrical insulation, the need to run the device in its real operating environment without access to debug ports etc. In such cases the built-in reporting of C-RUN messages can be tailored to suit your needs. The final output of the messages can be overridden to store them on disk, output them on a serial port or store them by any means available. The project from the previous example is prepared to show how the reporting can be changed to accommodate.

1. Choose the project from the previous example and make sure that the selected build configuration is *CheckPointersFromNonInstrumentedCode.* Right-click on the file *ReportCheckFailedStdout.c* in the project browser and select *Options…*

   Make sure that the file is not excluded from build and that the bounds-checking options look like below:



(continued on next page)

Using your own reporting

2. Re-open the options dialog on project level and in the Linker category make sure that the *Use command line options* check box on the *Extra options* tab is checked. There is already an extra command line option filled in to replace the standard reporting module with the file in the project.

3. Build and run the program in C-SPY. Notice how the error in this case comes out on *stdout* mixed with the other output from the program. You can feed all the output into *cspybat*, the batch processing debugger utility, and get a cleartext representation with basically the same content as what's shown in the C-RUN message window.

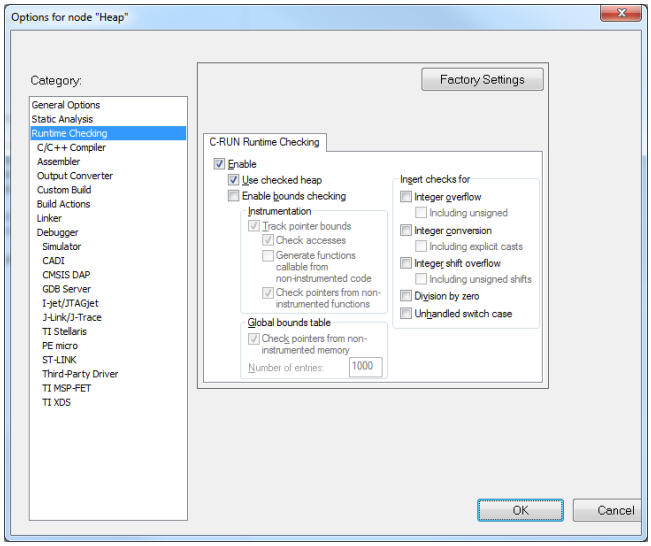
   Read more about how to use *cspybat* for message filtering in the *C-SPY debugging guide*.

4. Look inside *ReportCheckFailedStdout.c.* There is mainly formatting of hexadecimal numbers going on. The actual output to *stdout* can be replaced with any output method.

Heap checking

In applications using dynamically allocated memory it can happen that the application uses already freed memory, fail to de-allocate not needed anymore or simply writes outside allocated memory. A *special debug heap* and associated checker functions can be used to track down. Heap checking is based on the idea that each memory block is expanded with bookkeeping information and a buffer area, so that the blocks as seen by the application are not located side by side. The various checker functions examine the bookkeeping information and the buffer areas and can report violations of correct heap usage.

Note 1: The debug heap *increases* the likelihood to find heap usage errors, but it's not failsafe.
Note 2: Due to different performance and overhead characteristics for heap checking and bounds checking, the two methods can complement each other in tracking down dynamic memory usage errors, but it's not necessarily a good idea to enable them at the same time.



1. Choose the *Heap project* and make sure that *Use checked heap* is enabled like in the picture above. Build and run the program in C-SPY. For each reported error, examine the comments at that location in the code.

2. Comment out the line *listsize = __iar_set_delayed_free_size(2);* and re-build and run

3. For more in-depth information, read the section *Detecting heap memory leaks* and *Detecting heap integrity violations* in the *C-SPY Debugging Guide*.

IAR SYSTEMS

Questions?

www.iar.com/contact

Request a quote
for a standard license
www.iar.com/buy