Code Analysis Made Easy

ARMTechCon
Rafael Taubinger – Sr. Field Application Engineer
AGENDA

- Code Analysis Introduction
- Static Analysis with C-STAT
- Runtime Analysis with C-RUN
- Demo
- Summary
- Q&A
Code Analysis Tools
Introduction
WHY DO WE NEED ANALYSIS TOOLS?

1. C is not safe
2. All software contains bugs
3. The later you find a bug, the more expensive it gets
• Code analysis tools – Detecting erroneous code in the application
  • Static analysis tools
    • Analyze the source code without executing them – C-STAT
  • Runtime analysis tools
    • Analyze the source code dynamically during execution - C-RUN
Static Analysis with C-STAT
WHAT IS C-STAT?

- Complete static analysis tool
- Fully integrated in IAR Embedded Workbench
- Support for export/import of selected checks
- List of messages and data base file available
- Support for command line execution
- Includes:
  - Standard checks
    - More than 200 additional checks to address issues covered by CWE (the Common Weakness Enumeration), the CERT C coding standard etc.
    - Categories like Array bounds, Arithmetic errors
  - MISRA-C: 2004
  - MISRA-C: 2012
    - Improved version of the 2004 edition
    - Support for C99 ANSI-C standard, as well as C90
    - More precise checks
  - MISRA-C++: 2008
    - 165 checks to address C++ specific issues
• Import and Export settings
- Run code analysis for whole project or a specific file
ANALYZE THE RESULTS

- List of messages for every file
• Select the message and press F1 for the context sensitive help
• Full HTML report
• List of messages also available in a data base file
• Single file, e.g.:
  ```
  icstat --db a.db --checks checks.ch analyze -- iccarm cstat1.c
  ```

  Results

  selection of rules

• Multiple files, e.g.:
  ```
  icstat --db a.db --checks commands commands.txt
  ```

  analyze -- iccarm compiler_opts file1.c
  analyze -- iccarm compiler_opts file2.c
  link_analyze -- ilinkarm linker_opts file1.o file2.o

  Commands.txt
Runtime Analysis with C-RUN
WHAT IS C-RUN?

• C-RUN is a runtime analysis tool developed by IAR Systems
  • Both C and C++ source code are supported

• C-RUN is an add-on product of IAR Embedded Workbench
  • Fully integrated
  • No additional installation
  • Cannot work with 3rd-party compiler & debugger tools

• Target support
  • All ARM cores are supported
Traditional runtime analysis tools:
- Independent with compiler and debugger;
- Different applications and license models;
- Less knowledge about the target and optimization;
- Insert test code at the source code level;
- Large overhead in target memory and speed performance.

C-RUN:
- Created by compiler and debugger experts;
- Fully integrated within IAR Embedded Workbench;
- Insert target optimized test code directly during compilation;
- Replace the C/C++ standard library with a dedicated library which contains special functionality for runtime error checking;
- Result in minimized ROM/RAM overhead and speed penalty.
• Arithmetic checking
  - Integer overflows
  - Value-changing conversions
  - Shift overflows
  - Division by zero
  - Unhandled cases in switch statements

• Bounds checking
  - Accesses outside the bounds of arrays and other objects

• Heap checking
  - Out of heap space
  - Non-matching allocation and deallocation
  - Incorrect write accesses to heap memory
  - Memory leaks
C-RUN in IAR Embedded Workbench

Heap checking

Bounds checking

Arithmetic checking

- Enable
- Use checked heap
- Enable bounds checking
- Instrumentation:
  - Track pointer bounds
  - Check accesses
  - Generate functions callable from non-instrumented code
  - Check pointers from non-instrumented functions

Insert checks for:
- Integer overflow
- Integer conversion
- Integer shift overflow
- Division by zero
- Unhandled switch case

Global bounds table
- Check pointers from non-instrumented memory
- Number of entries: 1000
void main (void)
{
    int v1, v2, v3;

    v1 = 5;
    v2 = 0;
    v3 = v1/v2;  /* division by zero */
}

Division by zero
EXAMPLE 2: UNHANDLED SWITCH-CASE

```c
void main (void) {
    int i;

    for (i=0; i<2; i++)
    {
        switch (i) /* case 1 is not handled */
        {
            case 0:
                break;
            }
    }
}
```

![Image showing a compiler warning for an unhandled switch case](image-url)
EXAMPLE 3: OUT-OF-BOUNDS

```c
int main (void)
{
    int i, j;
    int a[3] = {1, 2, 3};
    for (i=0; a[i]!=0; i++) /* out of bounds */
    {
        /* when i==3 */
        j = a[i];
    }
    return j;
}
```
• Control how C-RUN messages are reported and what actions to be taken.
• Right-click on a C-RUN message and add a rule from the context menu:
  • Add Rule for ‘...’ at ‘range’
    • Add a rule that matches this particular runtime check at this particular location.
  • Add Rule for ‘...’ in ‘filename’
    • Add a rule that matches all runtime checks of this kind in the specified source file.
• Add Rule for ‘...’
  • Add a rule that matches all runtime checks of this kind in the whole application.
• Arithmetic checking
  • Compiler inserts instrumentation code to perform the check.
  • Code size increases (slightly).

• Bounds checking
  • Involves instrumentation code to track the bounds of a pointer when it is initialized, transferred, stored, passed to or returned from a function.
  • A global bounds table is required for indirectly accessed pointers.
  • Both code/data size and execution speed can increase (relatively high).

• Heap checking
  • The “checked heap” (provided by a dedicated runtime library) will replace the normal heap for the whole application.
  • Increases the memory size for each allocated heap block.
  • Heap operations can take longer time than with the normal heap.
Control of individual checks
- Turns on each C-RUN check individually, instead of enabling all checks.

Module based configuration
- Turns on C-RUN checks for specified modules, instead of whole application.
- Some checks must be set globally:
  - Use checked heap
  - Enable bounds checking
  - Global bounds table

Shrink the size of global bounds table
- In case that the application does not use too many indirectly accessed pointers.

Function based control
- #pragma disable_check
- #pragma no_bounds

configurable on module basis
C-RUN IN STAND-ALONE MODE

- Redirect the C-RUN messages to Terminal I/O
- Redirect C-RUN messages to UART
- Redirect C-RUN messages to RAM/SRAM
- Parse the C-RUN raw data with cspybat.exe
C-STAT AND C-RUN WITH ECLIPSE PLUGIN
Demo
• C-STAT and C-RUN are fully integrated within IAR Embedded Workbench Easy configuration and possibility to reuse the same setup on other projects.
• Data base file of list of messages available in case of certification
• C-STAT helps maintaining general code quality and portability by enforcing coding standards like MISRA C:2012 and others
  • Helps avoiding the pitfalls of C/C++ and to steer clear of implementation defined behavior as well as undefined behavior
• C-RUN performance
  • Minimum memory overhead and performance penalty.
  • Enable runtime error checking in resource constrained targets.
• C-STAT and C-RUN feature-rich
  • Discover common vulnerabilities in C/C++ programming language.
  • Fine-grained control of individual checks and specific code location.
  • Flexible error filter management and action control.
Want to learn more?
- Book an IAR Academy course
- Get a demo of our latest news

THANK YOU FOR YOUR ATTENTION!