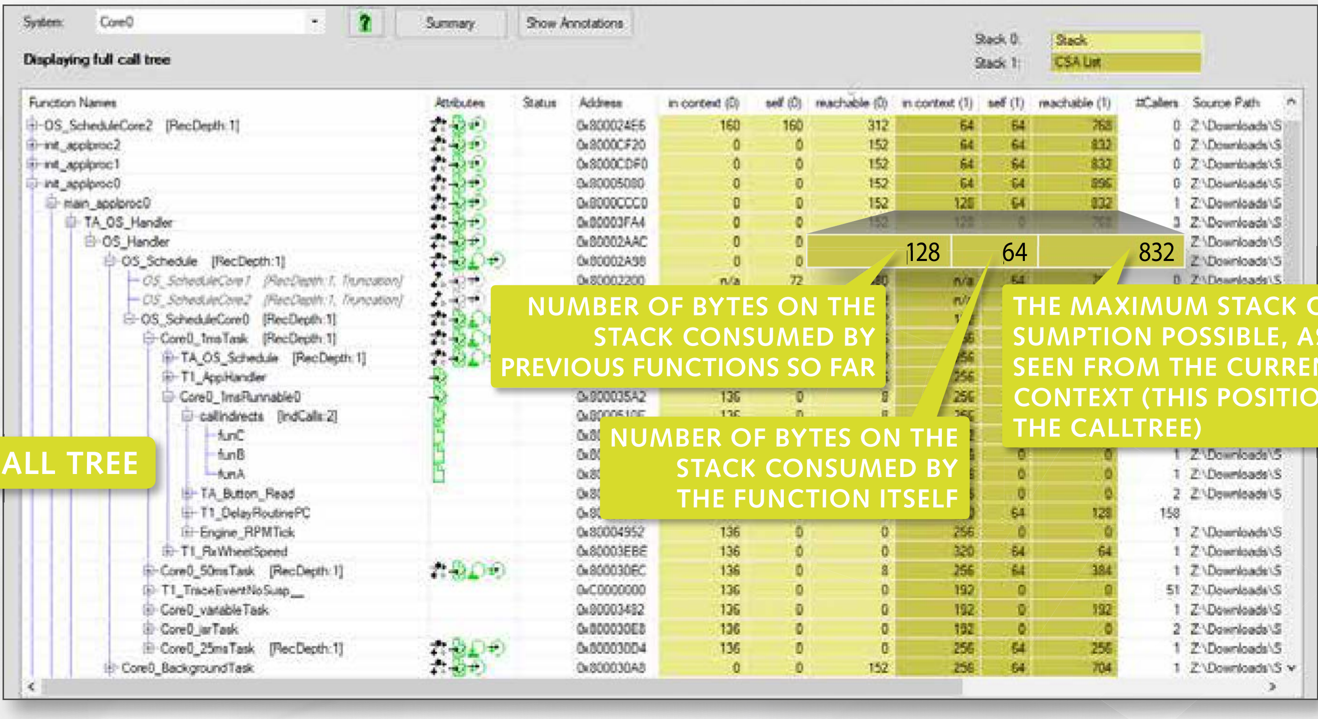
T1.stack

COST-EFFECTIVE, DETAILED AND ACCURATE STACK ANALYSIS

Introduction

Static stack analysis with T1.stack

Based on the binary (the ELF file), **T1.stack** performs a static code analysis: the binary is disassembled, function calls are extracted and the call tree is reconstructed. At the same time the stack consumption for each function is determined. The call tree and the stack consumption per function are combined into the comprehensive and powerful **T1.stack** view.



Indirect function calls

For any static code analysis there are limitations with respect to resolving indirect function calls. Such calls typically use function pointers and it is essential to know all call-targets (functions) which can possibly be called at run-time. **T1.stack** allows to complete any gaps in the static analysis **through annotation**. Three kinds of annotation are supported: manual annotation, import of generated annotation files and annotation through T1.flex measurements. Simply **measuring** call-targets is unique and a highlight of **T1.stack**. Such measurements can also be used to cross-check and verify annotations from other sources.

See example on the back page:

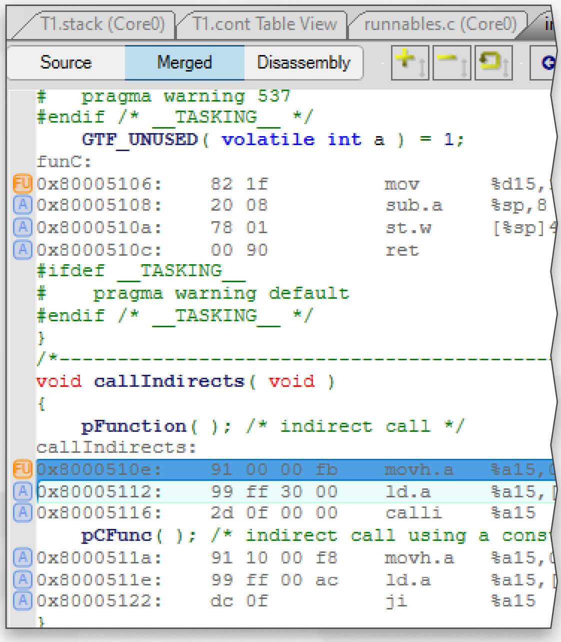
Unresolved indirect function call:

Core0_1msRunnable0	0x800035A2	136	0	8	256	64	192
callIndirects [IndCalls:2]	0x8000510E	136	0	8	256	64	0
funC	0x80005106	144	8	8	192	0	0
TA_Button_Read	0x8000400C	136	0	0	256	0	0
T1_DelayRoutinePC	0x80008F8E	136	0	0	320	64	128
Engine_RPMTick	0x80004952	136	0	0	256	0	0

Resolved indirect function call by dynamic T1.flex measurement:

Core0_1msRunnable0	0x800035A2	136	0	8	256	64	192
callIndirects [IndCalls:2]	0x8000510E	136	0	8	256	64	64
funC	0x80005106	144	8	8	192	0	0
funB	0x800050F4	136	0	0	256	0	0
funA	0x800050E2	136	0	0	256	0	0
TA_Button_Read	0x8000400C	136	0	0	256	0	0
T1_DelayRoutinePC	0x80008F8E	136	0	0	320	64	128
Engine_RPMTick	0x80004952	136	0	0	256	0	0

T1.stack offers the advantage of detailed analysis. It detects not only of the amount of used stack but also how and why it is used. Deep understanding of stack consumption allows successful optimization of stack usage and detection of unintended or purposeless use of the stack. Using less stack often helps to improve runtime performance.
When using a high level language it is not possible to predict the stack usage from even a detailed knowledge of the C source code. With auto-generated code, the problem is even worse. Using **T1.stack**, stack consumption can be continually tracked so that the effects of coding and compiler flags can be monitored and understood.
The accurate and detailed analysis of total stack usage combined with validation allows stacks to be reliably dimensioned with T1.stack and thus avoids the waste of allocating unnecessary memory. What's more: stack-overflows can be avoided.



Key benefits include:

- Static analysis based on the **binary file**
- 3rd party code can be **analyzed without the source code**
- **Compiler effects** (e.g. optimizations) are also taken into account
- **Measurement assisted resolving** of indirect function calls (function pointers)
- **Extreme fast analysis** (e.g. a 150MB ELF file of an engine management ECU could be analyzed in less than two minutes on a regular PC)
- **Call tree offers additional insights** into the software structure
- Built-In **source code- and disassembly-viewer**

Technical data

Supported CPU architectures:

- Infineon TriCore (*)
- ARM, ARM Thumb
- PowerPC, PowerPC VLE (*)
- RH850
- x86

(*) enhanced static analysis engine

GLIWA GmbH embedded systems
Pollinger Str. 1
82362 Weilheim i.OB. | Germany

gliwa.com

fon +49 - 881 - 13 85 22 - 0
fax +49 - 881 - 13 85 22 - 99
mail: info@gliwa.com