TIMING SUITE FOR REAL-TIME SYSTEMS

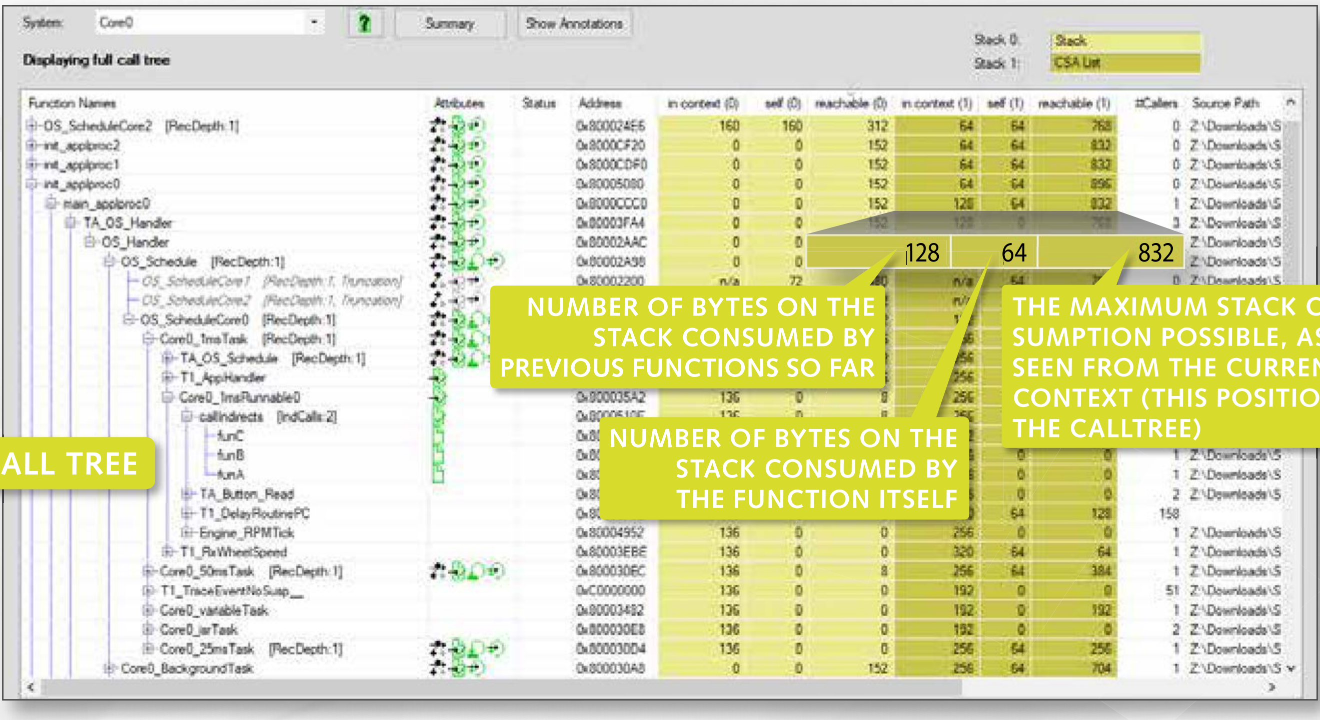
T1.stack

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.


CALL TREE
128
64
832
NUMBER OF BYTES ON THE STACK CONSUMED BY PREVIOUS FUNCTIONS SO FAR
NUMBER OF BYTES ON THE STACK CONSUMED BY THE FUNCTION ITSELF
THE MAXIMUM STACK CONSUMPTION POSSIBLE, AS SEEN FROM THE CURRENT CONTEXT (THIS POSITION IN THE CALLTREE)
System:
Summary
Show Annotations

System: Core0

Displaying full call tree

Stack 0: Stack
Stack 1: CSA Lim

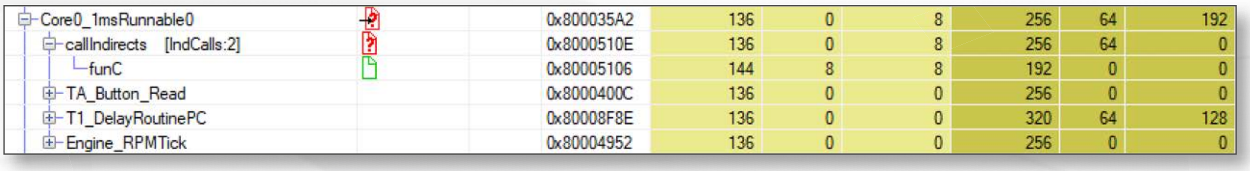
Function Names	Attributes	Status	Address	in context (0)	self (0)	reachable (0)	in context (1)	self (1)	reachable (1)	#Callers	Source Path
OS_ScheduleCore2 [RecDepth: 1]			0x800024E6	160	160	312	64	64	768	0	Z:\Downloads\S
init_applproc2			0x8000CF20	0	0	152	64	64	832	0	Z:\Downloads\S
init_applproc1			0x8000CDF0	0	0	152	64	64	832	0	Z:\Downloads\S
init_applproc0			0x80005080	0	0	152	64	64	896	0	Z:\Downloads\S
main_applproc0			0x8000CCC0	0	0	152	128	64	832	1	Z:\Downloads\S
TA_OS_Handler			0x80003FA4	0	0	152	128	0	768	3	Z:\Downloads\S
OS_Handler			0x80002AAC	0	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Z:\Downloads\S
OS_Schedule [RecDepth: 1]			0x80002A98	0	0	[illegible]	[illegible]	[illegible]	[illegible]	0	Z:\Downloads\S
OS_ScheduleCore1 [RecDepth: 1, Truncation]			0x80002200	n/a	72	[illegible]	n/a	64	[illegible]	0	Z:\Downloads\S
OS_ScheduleCore2 [RecDepth: 1, Truncation]			[illegible]	[illegible]	[illegible]	[illegible]	n/a	[illegible]	[illegible]	[illegible]	[illegible]
OS_ScheduleCore0 [RecDepth: 1]			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Core0_1msTask [RecDepth: 1]			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
TA_OS_Schedule [RecDepth: 1]			[illegible]	[illegible]	[illegible]	[illegible]	256	[illegible]	[illegible]	[illegible]	[illegible]
T1_AppHandler			[illegible]	[illegible]	[illegible]	[illegible]	256	[illegible]	[illegible]	[illegible]	[illegible]
Core0_1msRunnable0			0x800035A2	136	0	8	256	[illegible]	[illegible]	[illegible]	[illegible]
callIndirects [IndCalls: 2]			[illegible]	136	0	8	256	[illegible]	[illegible]	[illegible]	[illegible]
funC			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
funB			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	1	Z:\Downloads\S
funA			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	1	Z:\Downloads\S
TA_Button_Read			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	2	Z:\Downloads\S
T1_DelayRoutinePC			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	64	128	158	
Engine_RPMTick			0x80004952	136	0	0	256	0	0	1	Z:\Downloads\S
T1_RxWheelSpeed			0x80003EBE	136	0	0	320	64	64	1	Z:\Downloads\S
Core0_50msTask [RecDepth: 1]			0x800030EC	136	0	8	256	64	384	1	Z:\Downloads\S
T1_TraceEventNoSusp__			0xC0000000	136	0	0	192	0	0	51	Z:\Downloads\S
Core0_variableTask			0x80003482	136	0	0	192	0	192	1	Z:\Downloads\S
Core0_isrTask			0x800030E8	136	0	0	192	0	0	2	Z:\Downloads\S
Core0_25msTask [RecDepth: 1]			0x800030D4	136	0	0	256	64	256	1	Z:\Downloads\S
Core0_BackgroundTask			0x800030A8	0	0	152	256	64	704	1	Z:\Downloads\S

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.

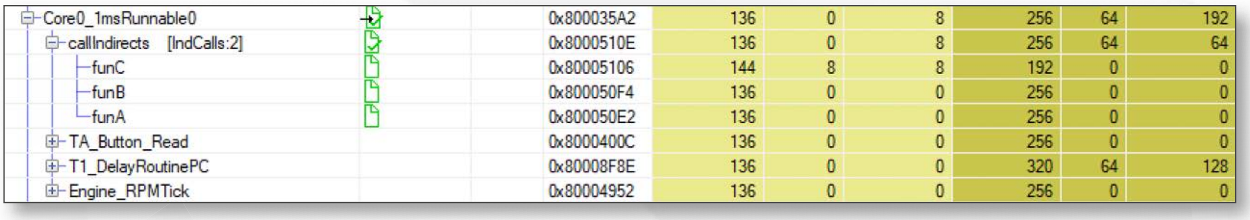
See example on the back page:

Unresolved indirect function call:



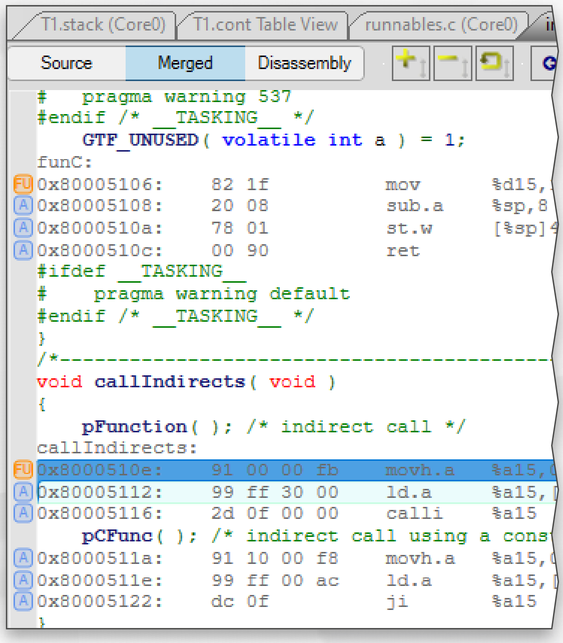
Core0_1msRunnable0	0x800035A2	136	0	8	256	64	192
callIndirects [IndCalls:2]	0x8000510E	136	0	8	256	64	0
funC	0x80005106	144	8	8	192	0	0
TA_Button_Read	0x8000400C	136	0	0	256	0	0
T1_DelayRoutinePC	0x80008F8E	136	0	0	320	64	128
Engine_RPMTick	0x80004952	136	0	0	256	0	0

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.



```
#   pragma warning 537
#endif /* __TASKING__ */
    GTF_UNUSED( volatile int a ) = 1;
funC:
0x80005106:    82 1f          mov      %d15,
0x80005108:    20 08          sub.a    %sp,8
0x8000510a:    78 01          st.w     [%sp]4
0x8000510c:    00 90          ret
#ifdef __TASKING__
#    pragma warning default
#endif /* __TASKING__ */
}
/*-------------------------------------------
void callIndirects( void )
{
    pFunction( ); /* indirect call */
callIndirects:
0x8000510e:    91 00 00 fb    movh.a   %a15,
0x80005112:    99 ff 30 00    ld.a     %a15,
0x80005116:    2d 0f 00 00    calli    %a15
    pCFunc( ); /* indirect call using a cons
0x8000511a:    91 10 00 f8    movh.a   %a15,
0x8000511e:    99 ff 00 ac    ld.a     %a15,
0x80005122:    dc 0f          ji       %a15
}
```

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