

∞ Infinity- Embedded Memory Usage and Profiling Tool that Supports 10 Cross-Tool Chains ∞

Infinity - Madonna Embedded Memory Usage and Profiling Tool

File Cross-Toolchains Calculator

> Tricore_tc21x_Project
> aarch64_unknown_elf_a
> arm_100
> arm_101
> arm_102
> arm_104
> arm_15789
> arm_199
> arm_beta_release
> Arm_CortexM3_Testing
> arm_csv_02
> arm_csv_test
> arm_mem_layout_105
> Arm_mem_layout_4555:
> Arm_Mem_Sum_Table
> Arm_none_Tl_cgt_test
> Arm_Not_Stripped
> Arm_Pyd_098976
> arm_pyd_23895
> arm_pyd_test666
> Arm_pyd_test_05
> Arm_pyd_test_09
> arm_pyd_test_105
> arm_pyd_test_57689
> Arm_Stripped
> arm_sum_table_796
> Arm_test_105678
> arm_Test_1076
> arm_test_10987
> arm_test_11234
> arm_test_pyd
> Axf_Format_Testing
> Axf_Format_Testing_v2
> AXF_Test_04
> c2000_layout_5054
> c2000_layout_600

Tricore

Enter text to search...

∞ Welcome To Infinity Memory Usage and Profiling Tool ∞

Infinity supports 10 cross-toolchains in the embedded industry:

- arm none eabi
- Tricore
- LLVM Embedded Tool Chain Arm
- ti cgt arm
- ti cgt armlvm
- aarch64 none elf
- ti cgt c2000
- ti cgt msp430
- powerpc
- Riscv32 Unknown Elf

Features of Infinity:

1. Charts for the memory usage of each memory region
2. Table summary for each memory region
3. Table summary of the size of each section
4. Table summary for the not allocated sections
5. Table summary for sections that have different virtual and physical addresses
6. Detects if there is a section overflow in the memory
7. Shows ELF headers
8. Draws memory layout for each memory region
9. Displays symbol table
10. Supports address-to-line feature
11. Displays assembly
12. Displays assembly of specific features
13. Generates report for linked and unlinked objects in memory
14. Strip ELF
15. Shows ELF headers and memory usage for stripped ELF
16. Doxygen documentation
17. Detects cyclic inclusion

Memory Allocation

Memory

Supported Toolchains



Supported Toolchains

- arm_none_eabi
- tricore
- LLVM_ET_Arm
- ti_cgt_arm
- ti_cgt_armlvm
- aarch64_none_elf
- ti_cgt_c2000
- ti-cgt-msp430
- powerpc
- riscv32_unknown_elf

∞ Why Infinity? ∞

Common Challenges in Memory Profiling and Embedded Development:

Memory Overflows: Hard-to-detect errors that lead to system instability or crashes, especially in embedded systems with limited memory.

Inefficient Memory Tracking: Difficulty in identifying memory consumption patterns across sections and regions, making optimization challenging.

Debugging Complexity: Time-consuming debugging due to limited insights into memory allocation and usage.

Limited Toolchain Support: Many tools lack compatibility with multiple toolchains, leading to inefficiencies in the development process.

Solution Offered by Infinity → How Infinity Addresses These Challenges:

Memory Layout Visualization: Offers a graphical representation of memory allocation, making issues like overflows immediately visible and easy to understand.

Comprehensive Memory Usage Analysis: Tracks memory consumption for each section, region, and symbol with detailed breakdowns.

Infinity generates tables such as:

Memory Usage Table: This table provides an overview of memory consumption across all sections, showing the total memory used and percentage usage for each section.

Section Usage Table: Displays detailed memory statistics per section (e.g., .text, .data, .bss), including used and free space, and any overflows detected.

Virtual and Physical Address Table: Lists sections that have different virtual and physical addresses, which is critical for understanding memory mapping.

Section Overflow Detection: Automatically identifies sections that exceed allocated memory, helping to prevent runtime errors and system crashes.

Toolchain Integration: Supports 10 cross-tool chains, ensuring seamless integration across diverse workflows.

Code Mapping: Maps memory addresses directly to lines of C code, simplifying the debugging process.

Infinity Features List:

1.Memory Layout Visualization

2.Comprehensive Memory Usage Analysis

3.Section Overflow Detection

4.Memory Usage Table (Displays memory consumption for each section and region)

5.Section Usage Table (Shows detailed memory usage statistics for each section, including free space and overflow information)

6.Virtual and Physical Address Mapping (Provides a table for sections with different virtual and physical memory addresses)

7.Toolchain Integration (Supports 10 cross-tool chains)

8.Code Mapping (Memory addresses to C code lines)

9.Documentation and Reports (Memory usage and memory layouts)

10.Symbol Table Generation

11.Assembly Generation for Specific Sections

12.Assembly Generation for All Sections

13.ELF Header Parsing

14.Stripping ELF (Optimizes file size for production)

15.Detecting Cyclic Inclusion (Identifies circular dependencies in headers)

16.Generating Doxygen Documentation (Automated, searchable code documentation)

17.Call Graphs (Visualize function call relationships)

18.Infinity as a Learning Tool (Interactive features to educate developers on memory profiling and best practices)

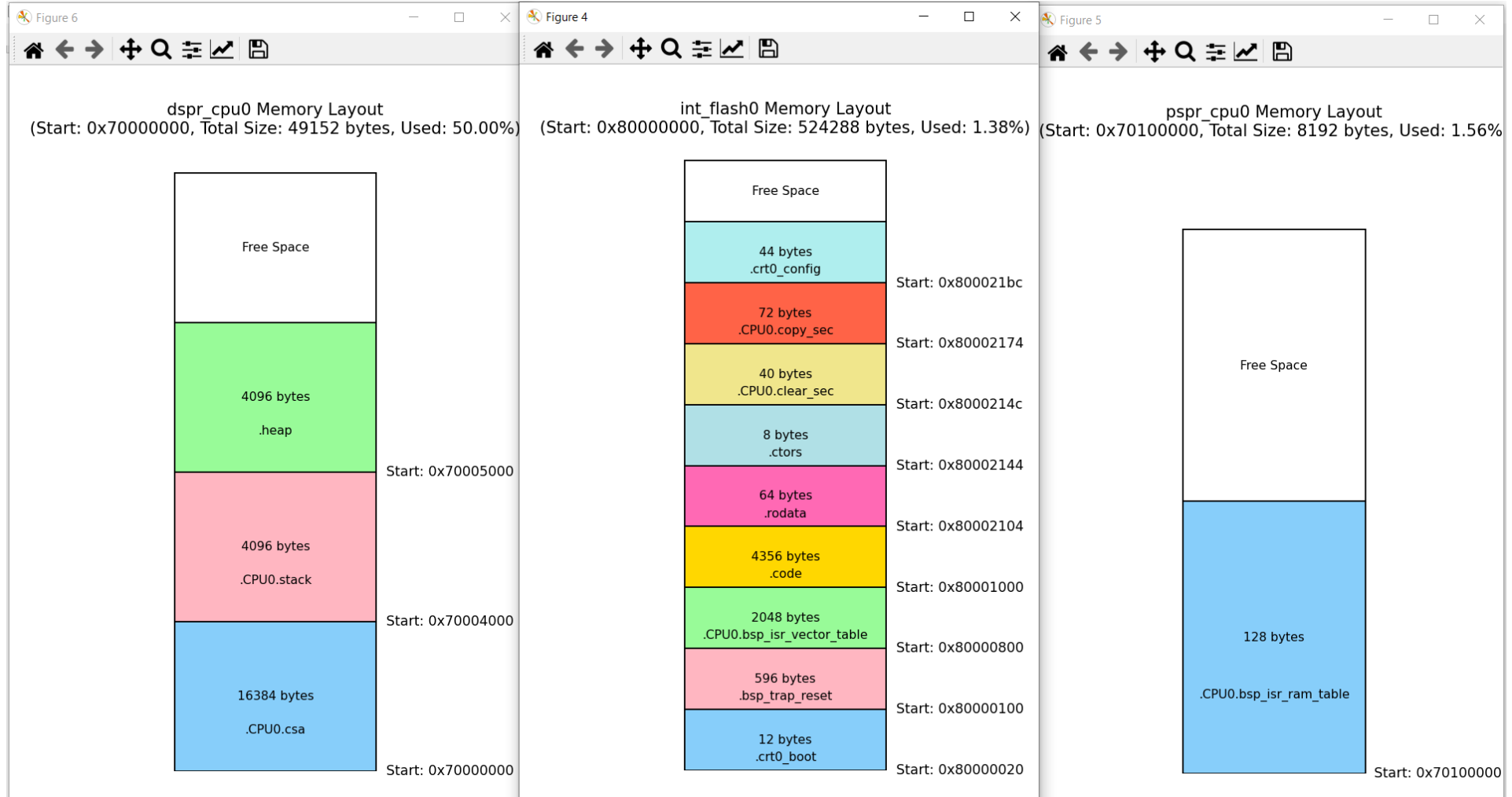
∞ Drawing Memory Layout Feature ∞

Simplifies Understanding of Memory Allocation:

Developers can easily understand which sections occupy which memory regions and how much space is left unused.

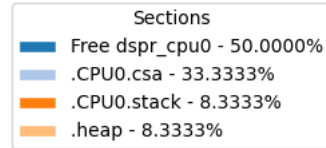
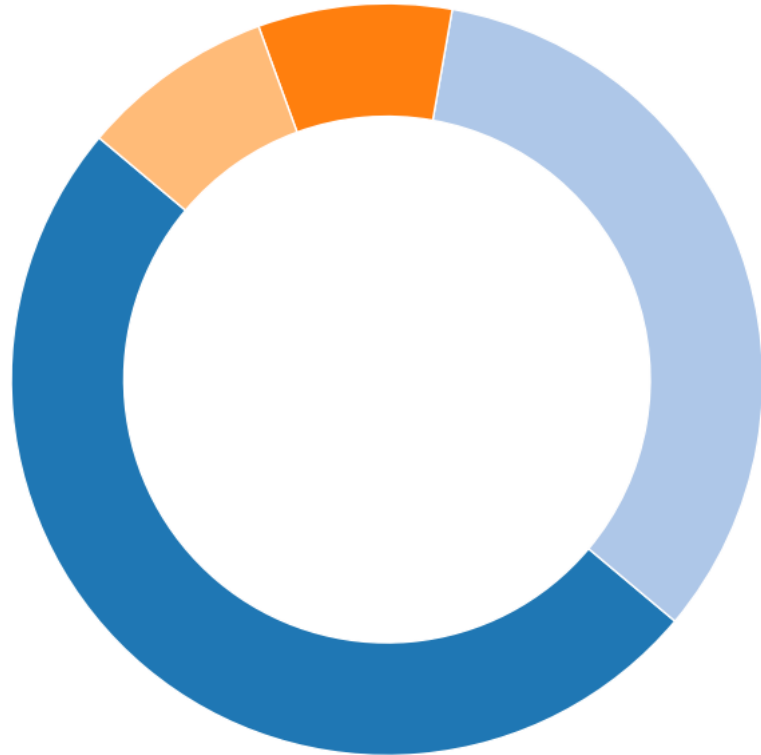
Identifies Underutilized or Overused Regions Quickly:

Overused memory regions can cause overflow, leading to instability or crashes, while underutilized regions indicate inefficient resource usage, potentially increasing costs. This feature helps developers quickly identify and address these issues by optimizing memory allocation or redistributing resources.

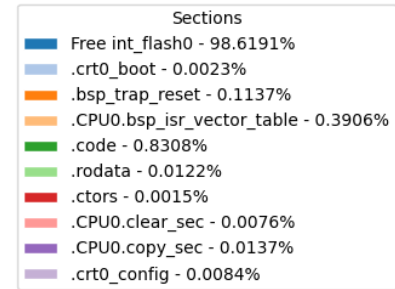
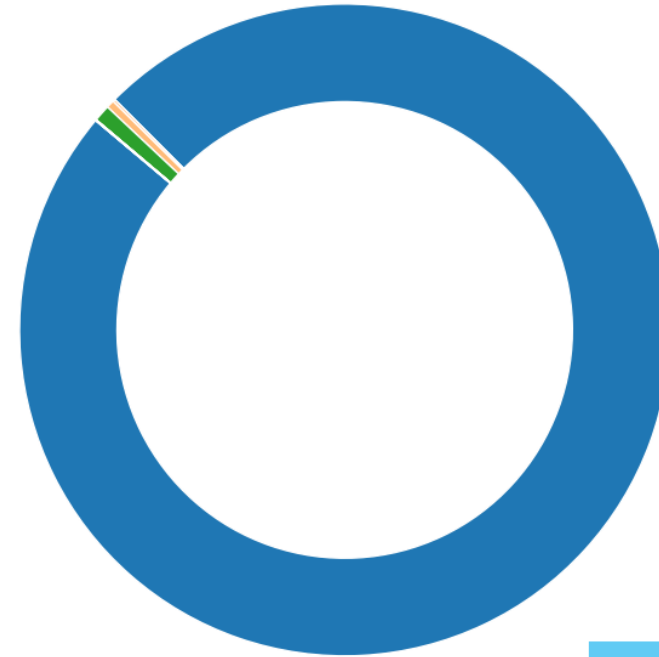


∞ Memory Usage Charts ∞

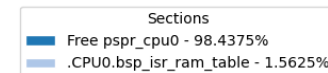
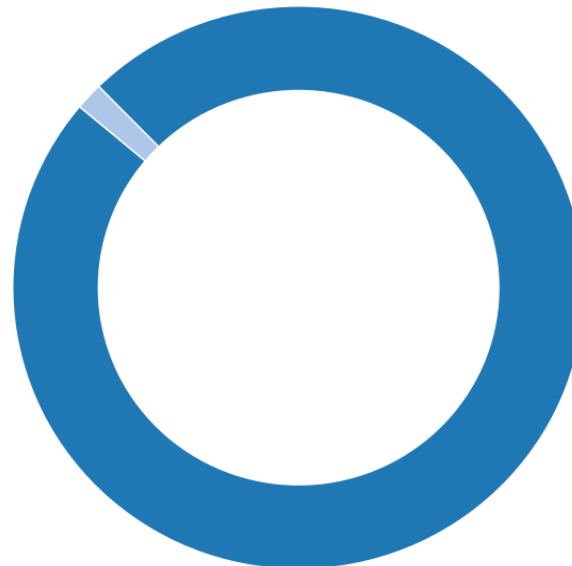
dspr_cpu0 Sections



int_flash0 Sections



pspr_cpu0 Sections



Provides a Quick, Visual Summary of Memory Utilization

Pie Charts: Represent each section as a slice, with the size of each slice proportional to the amount of memory consumed. This is useful for visualizing the proportion of each section relative to the total memory in a region.

∞ Memory Usage of each Section ∞

Section Name	Size (Bytes)	Size (Hex)	Virtual Address	Memory Type	Consumed %
.bmhd_0_org	32	0x20	0xa0000000	bmhd_0	100.0
.crt0_boot	12	0xc	0x80000020	int_flash0	0.0023
.bsp_trap_reset	596	0x254	0x80000100	int_flash0	0.1137
.CPU0.bsp_isr_vector_table	2048	0x800	0x80000800	int_flash0	0.3906
.CPU0.bsp_isr_ram_table	128	0x80	0x70100000	pspr_cpu0	1.5625
.CPU0.csa	16384	0x4000	0x70000000	dspr_cpu0	33.3333
.CPU0.stack	4096	0x1000	0x70004000	dspr_cpu0	8.3333
.code	4356	0x1104	0x80001000	int_flash0	0.8308
.rodata	64	0x40	0x80002104	int_flash0	0.0122
.ctors	8	0x8	0x80002144	int_flash0	0.0015
.heap	4096	0x1000	0x70005000	dspr_cpu0	8.3333
.CPU0.clear_sec	40	0x28	0x8000214c	int_flash0	0.0076
.CPU0.copy_sec	72	0x48	0x80002174	int_flash0	0.0137
.crt0_config	44	0x2c	0x800021bc	int_flash0	0.0084

∞ Consumed Percentage of each Memory Region ∞

Memory Name	attributes	origin	length (Bytes)	consumed size	Consumed %
int_flash0	rx	0x80000000	524288	7240	1.38092041015625
pspr_cpu0	rx	0x70100000	8192	128	1.5625
dspr_cpu0	w!x	0x70000000	49152	24576	50.0

bmhd_0	rx	0xa0000000	32	32	100.0
bmhd_1	rx	0xa0020000	32	0	0.0
bmhd_2	rx	0xa000ffe0	32	0	0.0
bmhd_3	rx	0xa001ffe0	32	0	0.0

∞ Sections Having Different Virtual and Physical Addresses ∞

This feature identifies and reports sections where the **virtual address (VA)** and **physical address (PA)** differ. It provides a detailed mapping of these addresses for each section.

The feature calculates the **memory consumption percentage** for both the virtual and physical memory of that section.

Validates Linker Script Configuration:

- Ensures that sections are mapped correctly as defined in the linker script, preventing runtime errors or inefficiencies caused by incorrect placement.

- Validates whether sections are using their allocated memory efficiently.

Section Name	Size (Hex)	Load Address	Runtime Address	Load Mem	Runtime Mem	% load mem	% runtime mem
.data	0xc	0x8002278	0x20000000	FLASH	RAM	0.0183	0.0586

Section Name	Size (Hex)	Load Address	Runtime Address	Load Mem	Runtime Mem	% load mem	% runtime mem
.data	0xc	0x8002278	0x20000000	FLASH	RAM	0.0183	0.0586

∞ Sections Not Allocated In Memory ∞

The "Sections Not Allocated in Memory" feature identifies sections that are intentionally excluded from the memory allocation.

These sections may include:

Debugging sections:

-Code and data related to debugging (e.g., symbol tables, debug information) that are not needed in production but are crucial during development.

Infinity - Madonna Embedded Memory Usage and Profiling Tool

File Cross-Toolchains Calculator

Tricore

Enter text to search...

Summary table saved to D:\Infinity_Work_Dir\Tricore_tc21x_Project\Sections_Not_Allocated_table.png

View Charts of Memory Allocation

Show Sections Not Allocated In Memory

Show sections having different virtual and physical addresses

Explain Section Overflow

Detect Section Overflow

Section Name	Size (Bytes)	Size (Hex)	Address
.comment	83	0x53	0x0
.debug_aranges	1080	0x438	0x0
.debug_info	62814	0x755e	0x0
.debug_abbrev	2433	0x981	0x0
.debug_line	21324	0x534c	0x0
.debug_frame	1400	0x578	0x0
.debug_str	642	0x282	0x0
.debug_loc	212	0xd4	0x0
.debug_ranges	528	0x210	0x0
.version_info	71713	0x11821	0x0
.debug_macro	1209317	0x1273e5	0x0

∞ Detecting section overflow ∞

Detecting Section Overflow feature monitors sections whose memory usage exceeds their allocated size. It analyzes each memory section, such as .text, .data, .bss, and .rodata, comparing the actual memory usage against the limits defined by the linker script or memory configuration.

Ensuring that each section stays within its allocated memory guarantees predictable and stable system behavior.

Developers can confidently deploy software knowing that memory-related issues, such as overflows, have been proactively addressed.

The screenshot displays the Infinity - Madonna Embedded Memory Usage and Profiling Tool interface. The left sidebar shows a project tree with a folder named 'TI_CGT_2000' expanded, listing various files like 'addr2line', 'Assembly', 'elf_all_Obj', etc. The main window is titled 'ti cgt c2000' and shows 'Overflow Details' for two sections: '.text part 1' and '.text part 2'. The details include memory regions, addresses, sizes, and consumed percentages. A table titled 'overflow_table.png' is overlaid on the bottom right, showing a summary of section overflow data.

Section Name	Total Size	Virtual Address	Memory Type	Consumed Size	Consumed %	Start Address	End Address
.text	0x266c	0009b398	FLASH_BANK0_SEC_104_111	0xc68	38.76953125	0x9b398	0x9bfff
.text	0x266c	0009b398	FLASHBANK0_CODE	0x3a04	60.472312703583064	0x9a000	0x9da03

∞ Symbol Table With Filtration of each Symbol Type ∞

The **Symbol Table with Filtration** feature enables developers to view and analyze the symbols within an ELF file or binary, categorized by their type. This functionality provides a filtered and organized view of symbols, making it easier to identify and work with specific categories.

Filtering by Symbol Size:

•Non-Zero Size Symbols:

Isolates objects or variables that occupy memory (e.g., arrays, structs).

•**Zero-Size Symbols:** Identifies placeholders or function declarations that don't consume memory but are still part of the symbol table.

Categorization by Symbol Type:

The **Symbol Table with Filtration** feature categorizes and filters symbols based on their **type** and **location within memory sections**, providing developers with a detailed and organized view.

The screenshot displays the 'Infinity - Madonna Embedded Memory Usage and Profiling Tool' interface. The 'Symbol Table' tab is active, showing a list of symbols with columns for Physical Address, Size, Type, and Object Name. The symbols are categorized by type, such as 'Score_id', 'Sfex', 'Sisp', 'Sfex', 'Spcxi', 'crt0_config', 'CSA_BASE_CPU0_', 'CTOR_END', 'CTOR_LIST', 'HEAP', 'HEAP_END', 'HEAP_SIZE', 'STACK_BASE_CPU0_', 'clear_exec_end', 'clear_exec_loop', 'copy_exec_end', 'copy_exec_loop', 'crt0_reset', 'ctor_exec_end', 'ctor_exec_loop', 'exit', 'SMALL_DATA', 'SMALL_DATA2', 'start', 'table_bss_clear_loop', 'table_bss_clear_loop_end', 'table_data_copy_loop', 'table_data_copy_loop_end', 'BMHD', 'board_led', 'bsp_board_led_Init', 'bsp_board_led_InitAll', 'bsp_board_led_Set', 'bsp_board_wdg_Disable', 'bsp_isr_Init', 'bsp_isr_RegisterHandler', 'BSP_ISR_VECTOR_TABLE_CPU0', 'BSP_TRAP_VECTOR_TABLE', 'uc_core_EnableICache', 'uc_core_GetCurrentCore', and 'uc_core_StartAllCores'.

On the right side, there is a 'Symbol Table' panel with several buttons for filtering and generating the symbol table:

- Generate Symbol Table
- Explain the type of each symbol
- Explanation of nm Utility and Linker relation
- Show All Symbols With Size Zero
- Show All Objects With a Non-zero Size
- Explain Why Zero object size found
- Show Tips And Tricks
- Filter Objects by their type ...

The 'Filter Objects by their type ...' dropdown menu is open, showing the following categories:

- Absolute (A)
- BSS (B/b)
- Common (C)
- Initialized Data (D/d)
- Small Data (G/g)
- DLL-specific (i)
- Indirect reference (I)
- Debugging (N)
- non-data, non-code, non-debug read-only section (n)

∞ Mapping the desired address to a specific line in c code ∞

This feature enables developers to **map a specific memory address back to the corresponding line of source code in a C file.**

It provides a seamless link between low-level memory details and the high-level code responsible for defining or using that memory location.

Key functionalities include:

- **Address-to-Code Mapping:**

The tool identifies the exact line of C source code corresponding to a given memory address.

- **Section and Memory Information:**

Alongside the source code mapping, the tool also provides: The memory section (e.g., .text, .data, .bss, .rodata) where the address resides. Additional details such as the start address, size, and attributes (e.g., writable, executable) of the section containing the specified address.

The screenshot displays the 'Infinity - Madonna Embedded Memory Usage and Profiling Tool' interface. The left sidebar shows a project tree for 'Tricore_tc21x_Project' with various files like 'addr2line.txt', 'Assembly_Output.tx', and 'elf_all_Objects.csv'. The main window is titled 'Tricore' and contains a search bar and a text area with the following content:

```
*****
All Information about Address 8000101a
*****

*****
Address location in memory:
*****
Address is located in section : .code
Address is located in memory : int_flash0

*****
About mapping address 8000101a to a specific line in c/c++ code
*****
Function name is : bsp_board_led_InitAll
Address 8000101a corresponds to line number 47 in file C:
\Users\DELL\htc-
workspace\tc21x_bsp_example\_iROM_TC21X_TRIBOARD_TC2X3_V1_1/./bsp/
board/bsp_board.c
```

Below the main window, a smaller window titled 'section and memory details for address 8000101a' is open, displaying the following information:

```
Section Name: .code
Section Start Address: 0x80001000
Section End Address: 0x80002103
Section Size: 0x1104

Memory Name: int_flash0
Memory Start Address: 0x80000000
Memory End Address: 0x8007ffff
Memory Size: 0x80000
```

On the right side of the interface, there is a tabbed menu with the following tabs: 'Unlinked Objects', 'Assembly Generation', 'Assembly of Specific Section', 'Address to line' (selected), 'Strip Elf', and 'Draw Memory Layout'. Below the tabs, there is a section titled 'Address to line' with three buttons:

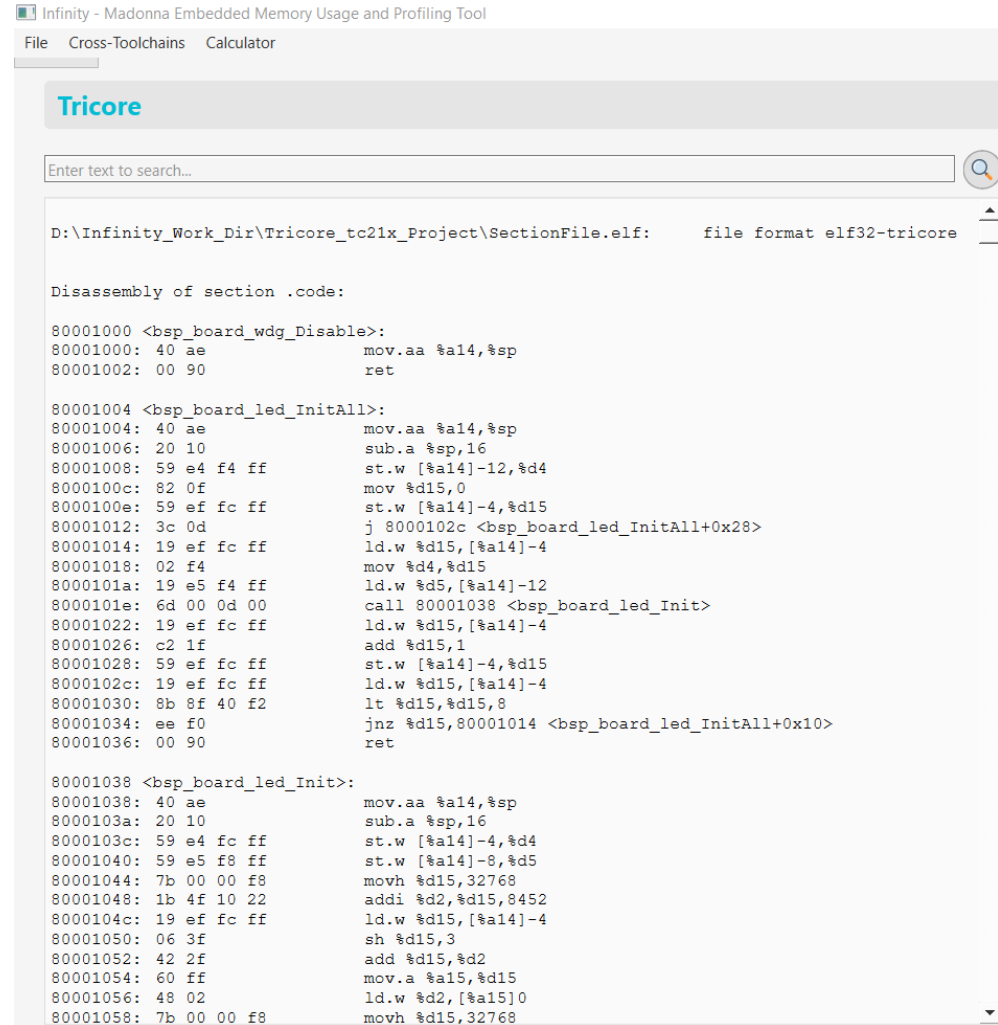
- Show the address corresponds to which line in C/C++
- More information about address to line conversion
- Why might the address-to-line utility fail to resolve an address?
- Show the section and memory details for that address

At the bottom right, there is a diagram illustrating the 'Address to line' process. It shows a flow from 'ADDRESS' to 'PROCESSING' (represented by a circular arrow) and then to 'LINE NUMBER IN C CODE'. The diagram is styled with a blue and white color scheme and includes circuit-like elements.

∞ Assembly Generation of a Specific Section ∞

Allowing developers to isolate and view the machine-level instructions of a chosen section rather than the entire assembly output.

Along with the assembly, the tool provides relevant ELF header information for the selected section, such as its address, size, alignment, and attributes, ensuring a comprehensive understanding of the section's role in the binary.



```
D:\Infinity_Work_Dir\Tricore_tc21x_Project\SectionFile.elf:    file format elf32-tricore

Disassembly of section .code:

80001000 <bsp_board_wdg_Disable>:
80001000: 40 ae          mov.aa %a14,%sp
80001002: 00 90          ret

80001004 <bsp_board_led_InitAll>:
80001004: 40 ae          mov.aa %a14,%sp
80001006: 20 10          sub.a %sp,16
80001008: 59 e4 f4 ff    st.w [%a14]-12,%d4
8000100c: 82 0f          mov %d15,0
8000100e: 59 ef fc ff    st.w [%a14]-4,%d15
80001012: 3c 0d          j 8000102c <bsp_board_led_InitAll+0x28>
80001014: 19 ef fc ff    ld.w %d15,[%a14]-4
80001018: 02 f4          mov %d4,%d15
8000101a: 19 e5 f4 ff    ld.w %d5,[%a14]-12
8000101e: 6d 00 0d 00    call 80001038 <bsp_board_led_Init>
80001022: 19 ef fc ff    ld.w %d15,[%a14]-4
80001026: c2 1f          add %d15,1
80001028: 59 ef fc ff    st.w [%a14]-4,%d15
8000102c: 19 ef fc ff    ld.w %d15,[%a14]-4
80001030: 8b 8f 40 f2    lt %d15,%d15,8
80001034: ee f0          jnz %d15,80001014 <bsp_board_led_InitAll+0x10>
80001036: 00 90          ret

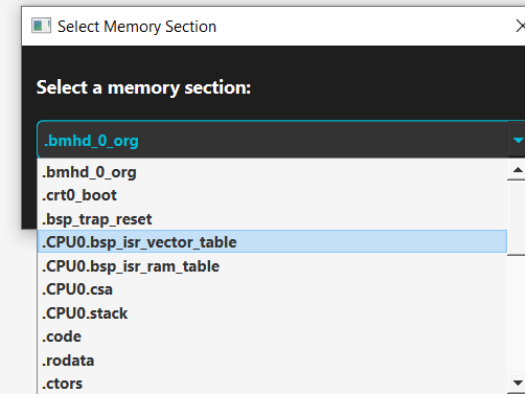
80001038 <bsp_board_led_Init>:
80001038: 40 ae          mov.aa %a14,%sp
8000103a: 20 10          sub.a %sp,16
8000103c: 59 e4 fc ff    st.w [%a14]-4,%d4
80001040: 59 e5 f8 ff    st.w [%a14]-8,%d5
80001044: 7b 00 00 f8    movh %d15,32768
80001048: 1b 4f 10 22    addi %d2,%d15,8452
8000104c: 19 ef fc ff    ld.w %d15,[%a14]-4
80001050: 06 3f          sh %d15,3
80001052: 42 2f          add %d15,%d2
80001054: 60 ff          mov.a %a15,%d15
80001056: 48 02          ld.w %d2,[%a15]0
80001058: 7b 00 00 f8    movh %d15,32768
```

Assembly of Specific Section

Generate Assembly of Specific Section

Why Some Data Sections Lack Disassembly Information

Show Elf Header Information Of That Section



∞ Assembly Generation of all Sections ∞

The **Assembly Generation of All Sections** feature generates a complete assembly code output for the entire project. Unlike the feature for specific sections, this provides a holistic view of all sections defined in the ELF file.

Developers can inspect every machine-level instruction, branch, and memory access in the program to identify problematic sequences or inefficiencies.

The screenshot displays the 'Infinity - Madonna Embedded Memory Usage and Profiling Tool' interface. On the left, a tree view lists various sections, with 'Tricore_tc21x_Prc' selected. The main window is titled 'Tricore' and shows a search bar with 'movh.a' entered. Below the search bar, the disassembly of several sections is visible, including '.bmhd_0_org', '.crt0_boot', and '.bsp_trap_reset'. The disassembly code is shown in a table format with columns for address, hex code, and assembly instructions. A 'Generate Assembly Code' button is prominently displayed on the right side of the interface. The bottom status bar indicates 'Match 1 of 46'.

Infinity - Madonna Embedded Memory Usage and Profiling Tool

File Cross-Toolchains Calculator

Tricore

movh.a

Previous Next

D:
\\gcc_arm_none_eabi\\Examples_Project\\tricore_ex2\\tc21x_bsp_example.elf:
file format elf32-tricore

Disassembly of section .bmhd_0_org:

```
a0000000 <BMHD>:  
a0000000: 20 00          sub.a %sp,0  
a0000002: 00 80          rfe  
a0000004: 70 03          .hword 0x0370  
a0000006: 59 b3 00 00    st.w [%a11]0,%d3  
...  
a0000016: 00 00          nop  
a0000018: b4 b5          st.h [%a11],%d5  
a000001a: e9 49 4b 4a    st.b [%a4]-23285,%d9  
a000001e: 16 b6          and %d15,182
```

Disassembly of section .crt0_boot:

```
80000020 <_crt0_reset>:  
80000020: 91 00 00 f8    movh.a %a15,32768  
80000024: d9 ff fa 61    lea %a15,[%a15]7610 <80001dba <_start>>  
80000028: 2d 0f 30 00    ji %a15
```

Disassembly of section .bsp_trap_reset:

```
80000100 <BSP_TRAP_VECTOR_TABLE>:  
80000100: 00 a0          debug  
80000102: 0d 00 00 02    svlxc  
80000106: 91 00 00 e8    movh.a %a14,32768  
8000010a: d9 ee 00 80    lea %a14,[%a14]512 <80000200  
<Trap_0_Handler>>  
8000010e: dc 0e          ji %a14  
...  
80000120: 00 a0          debug  
80000122: 0d 00 00 02    svlxc  
80000126: 91 00 00 e8    movh.a %a14,32768  
8000012a: d9 ee 1e 80    lea %a14,[%a14]542 <8000021e  
<Trap_1_Handler>>  
8000012e: dc 0e          ji %a14
```

Assembly Generation

Generate Assembly Code

Match 1 of 46

∞ Generating Documents For linked and unlinked objects ∞

The Generating Documents for Linked and Unlinked Objects

feature provides a detailed breakdown of all objects, categorized as either:

1.Linked Objects: objects that are successfully included in the final executable during the linking process.

2.Unlinked Objects: objects that were not included in the final executable, often due to build configurations.

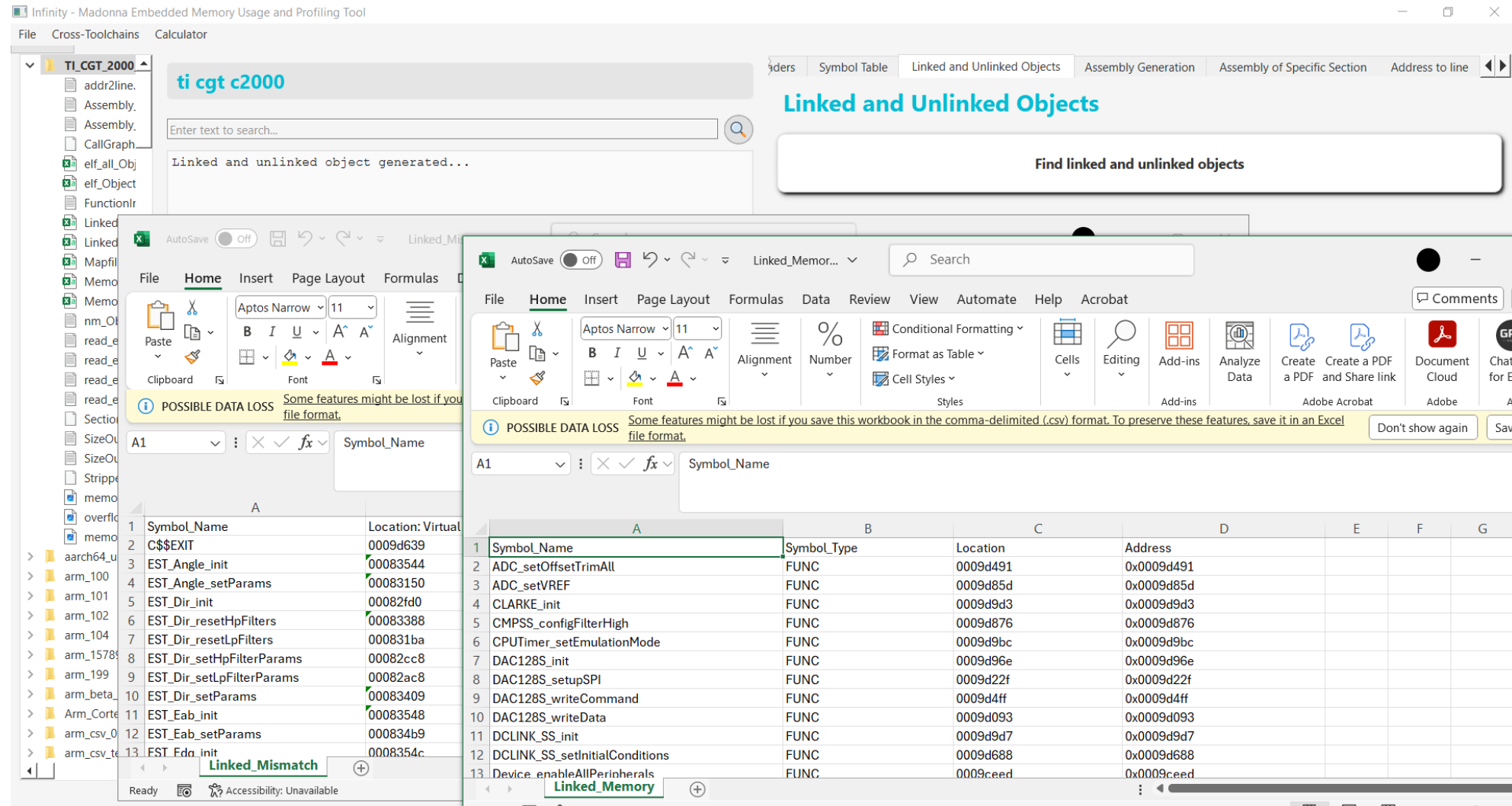
Provides Insights into Excluded Objects:

Objects that are excluded from the final build may represent: Unused Code: Functions or variables that are defined but never referenced.

Optional Features: Objects excluded due to conditional compilation or build settings.

By documenting unlinked objects, developers can:

Remove unnecessary objects to clean up the codebase.
Ensure that no critical objects are accidentally excluded.



Infinity - Madonna Embedded Memory Usage and Profiling Tool

File Cross-Toolchains Calculator

ti cgt c2000

Enter text to search...

Linked and unlinked object generated...

Find linked and unlinked objects

File Home Insert Page Layout Formulas

Clipboard Font Alignment

POSSIBLE DATA LOSS Some features might be lost if you save this workbook in the comma-delimited (.csv) format. To preserve these features, save it in an Excel file format.

Symbol_Name	Symbol_Type	Location	Address
C\$\$EXIT		0009d639	
EST_Angle_init	FUNC	00083544	0x0009d491
EST_Angle_setParams	FUNC	00083150	0x0009d85d
EST_Dir_init	FUNC	00082fd0	0x0009d9d3
EST_Dir_resetHpFilters	FUNC	00083388	0x0009d876
EST_Dir_resetLpFilters	FUNC	000831ba	0x0009d9bc
EST_Dir_setHpFilterParams	FUNC	00082cc8	0x0009d96e
EST_Dir_setLpFilterParams	FUNC	00082ac8	0x0009d22f
EST_Dir_setParams	FUNC	00083409	0x0009d4ff
EST_Eab_init	FUNC	00083548	0x0009d093
EST_Eab_setParams	FUNC	000834b9	0x0009d9d7
EST_Fda_init	FUNC	0008354c	0x0009d688
Device_enableAllPeripherals	FUNC	0009ceed	0x0009ceed

Ready Accessibility: Unavailable

∞ Elf Headers ∞

The ELF header is the starting point for understanding the organization and structure of an ELF file. By reading and validating the ELF header, developers can determine where data is located and how the sections and segments should be mapped when loaded into memory.

Infinity - Madonna Embedded Memory Usage and Profiling Tool

FileCross-ToolchainsCalculator

Tricore

Enter text to search...

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Lk	Inf	Al
[0]	NULL		00000000	000000	000000	00	0	0	0
[1]	.bshd_0_ory	PROGBITS	a0000000	0029e8	000020	00	0	0	1
	[00000006]:	ALLOC, EXEC							
[2]	.crt0_boot	PROGBITS	80000020	000820	00000c	00	0	0	1
	[00000006]:	ALLOC, EXEC							
[3]	.bsp_trap_reset	PROGBITS	80000100	000900	000254	00	0	0	256
	[00000006]:	ALLOC, EXEC							
[4]	.CPU0.bsp_isr_vector_table	PROGBITS	80000800	001000	000800	00	0	0	2048
	[00000006]:	ALLOC, EXEC							
[5]	.CPU0.bsp_isr_ram_table	NOBITS	70100000	0000f4	000080	00	0	0	4
	[00000003]:	WRITE, ALLOC							
[6]	.CPU0.csa	NOBITS	70000000	0000f4	004000	00	0	0	1
	[00000003]:	WRITE, ALLOC							
[7]	.CPU0.stack	NOBITS	70004000	0000f4	001000	00	0	0	1
	[00000003]:	WRITE, ALLOC							
[8]	.code	PROGBITS	80001000	001800	001104	00	0	0	2
	[00000006]:	ALLOC, EXEC							
[9]	.rodata	PROGBITS	80002104	002904	000040	00	0	0	4
	[00000002]:	ALLOC							
[10]	.ctors	PROGBITS	80002144	002944	000008	00	0	0	4
	[00000002]:	ALLOC							
[11]	.heap	NOBITS	70005000	002978	001000	00	0	0	8
	[00000003]:	WRITE, ALLOC							
[12]	.CPU0.clear_sec	PROGBITS	8000214c	00294c	000028	00	0	0	1

Memory AllocationMemory Table SummaryElf HeadersSymbol TableLinked and Unlinked ObjectsAssembly Generation

Elf Headers

Show detailed information of elf headers

∞ Stripping Elf ∞

The **Stripping ELF** feature is used to optimize ELF files by removing unnecessary sections and symbols that are not required for the execution of the program. This process reduces the file size, creating smaller, production-ready binaries while retaining only the essential components needed for runtime.

Ensures Security and Confidentiality:

- Debugging symbols and metadata often contain sensitive information about the program's structure and logic, such as variable names, function definitions, and source code mappings.
- Stripping removes this information, reducing the risk of reverse engineering.

The screenshot displays the Infinity - Madonna Embedded Memory Usage and Profiling Tool interface. The left sidebar shows a project tree for 'Tricore_tc21x_Project' with various files like 'addr2line.txt', 'Assembly_Output.tx', 'Assembly_SpecificSe', 'CallGraph.xml', 'elf_all_Objects.csv', 'elf_Objects.csv', 'FunctionInfo.txt', 'Linked_Memory.csv', 'Linked_Mismatch.cs', 'Mapfile_Parsing.csv', 'Memory_consumpti', 'Memory_consumpti', 'memory_regions.tal', 'memory_summary.t', 'nm_Objects.txt', 'read_elf.txt', 'read_elf_certain_sec', 'read_elf_section_to_', 'read_elf_stripped.txt', 'SectionFile.elf', 'Sections_Not_Alloca', 'SizeOutput.txt', 'SizeOutput_Stripped', and 'StrippedElf.elf'. The main window is titled 'Tricore' and contains a search bar and a list of ELF headers. The 'Strip ELF' tab is selected, showing a 'Strip ELF File' button and three options: 'Show Memory Usage Charts of stripped Elf', 'Show Memory Usage Table of stripped Elf', and 'Show Elf Headers of Stripped Elf'.

Infinity - Madonna Embedded Memory Usage and Profiling Tool

File Cross-Toolchains Calculator

Tricore

Enter text to search...

ELF Header:

```
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Infineon Tricore
Version: 0x1
Entry point address: 0x80000020
Start of program headers: 52 (bytes into file)
Start of section headers: 82776 (bytes into file)
Flags: 0x200000, TC1.6.1
Size of this header: 52 (bytes)
Size of program headers: 52 (bytes)
Number of program headers: 6
Size of section headers: 40 (bytes)
Number of section headers: 18
Section header string table index: 17
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Lk	Inf	Al
[0]	NULL		00000000	000000	000000	00	0	0	0
[1]	.bmmhd_0_org	PROGBITS	a0000000	0029e8	000020	00	0	0	1
[2]	.crt0_boot	PROGBITS	80000020	000820	00000c	00	0	0	1
[3]	.bsp_trap_reset	PROGBITS	80000100	000900	000254	00	0	0	256
[4]	.CPU0.bsp_isr_vector_table	PROGBITS	80000800	001000	000800	00	0	0	2048
[5]	.CPU0.bsp_isr_ram_table	NOBITS	70100000	0000f4	000080	00	0	0	4

Strip ELF

Strip ELF File

Show Memory Usage Charts of stripped Elf

Show Memory Usage Table of stripped Elf

Show Elf Headers of Stripped Elf

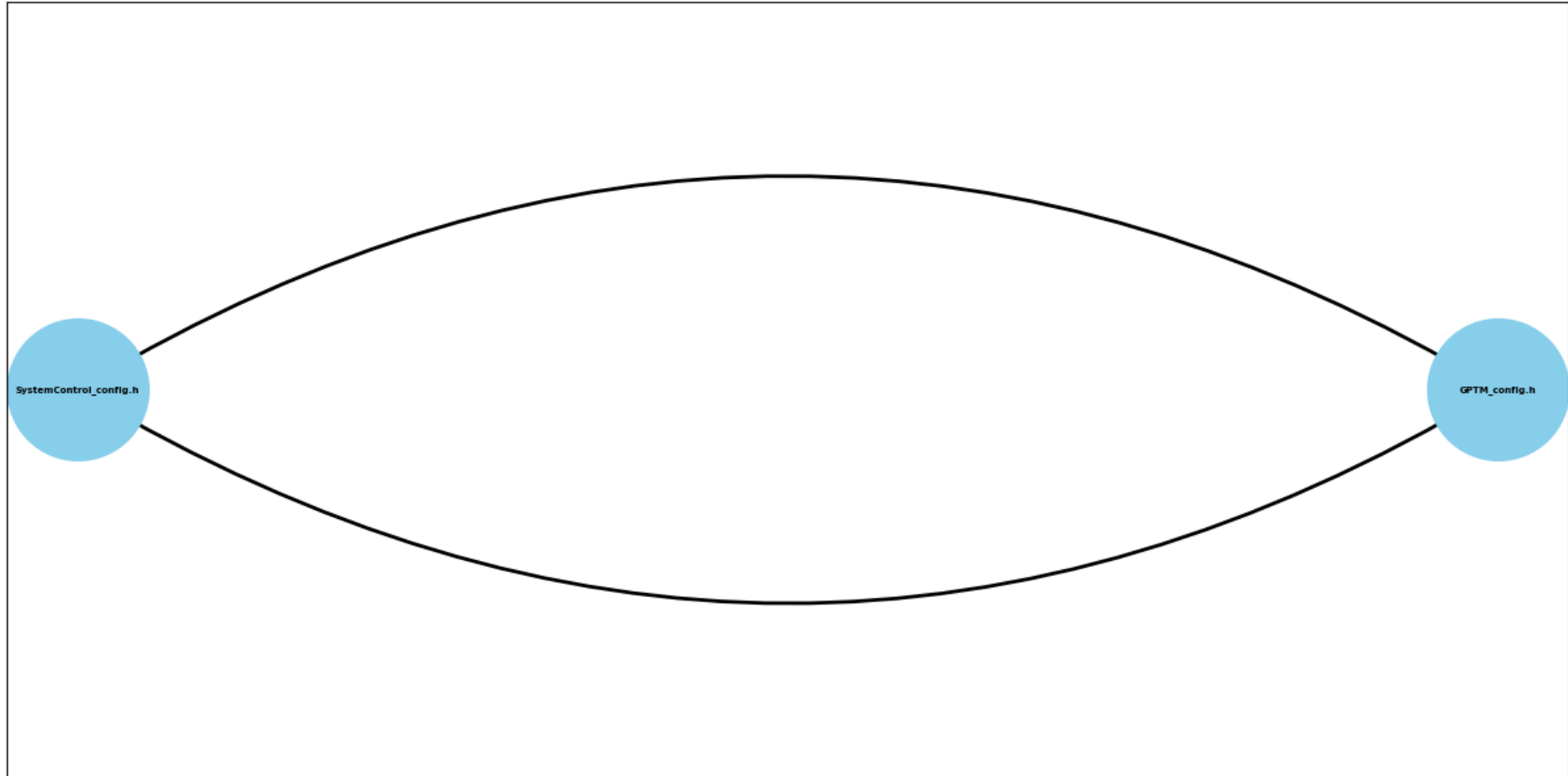
∞ Detecting Cyclic Inclusion ∞

The **Detecting Cyclic Inclusion** feature identifies and resolves **circular dependencies** in header files, modules, or source files. Cyclic inclusion occurs when two or more files reference each other directly or indirectly in a loop, leading to an **infinite inclusion cycle** during the preprocessing or compilation phase.

Dependency Analysis:

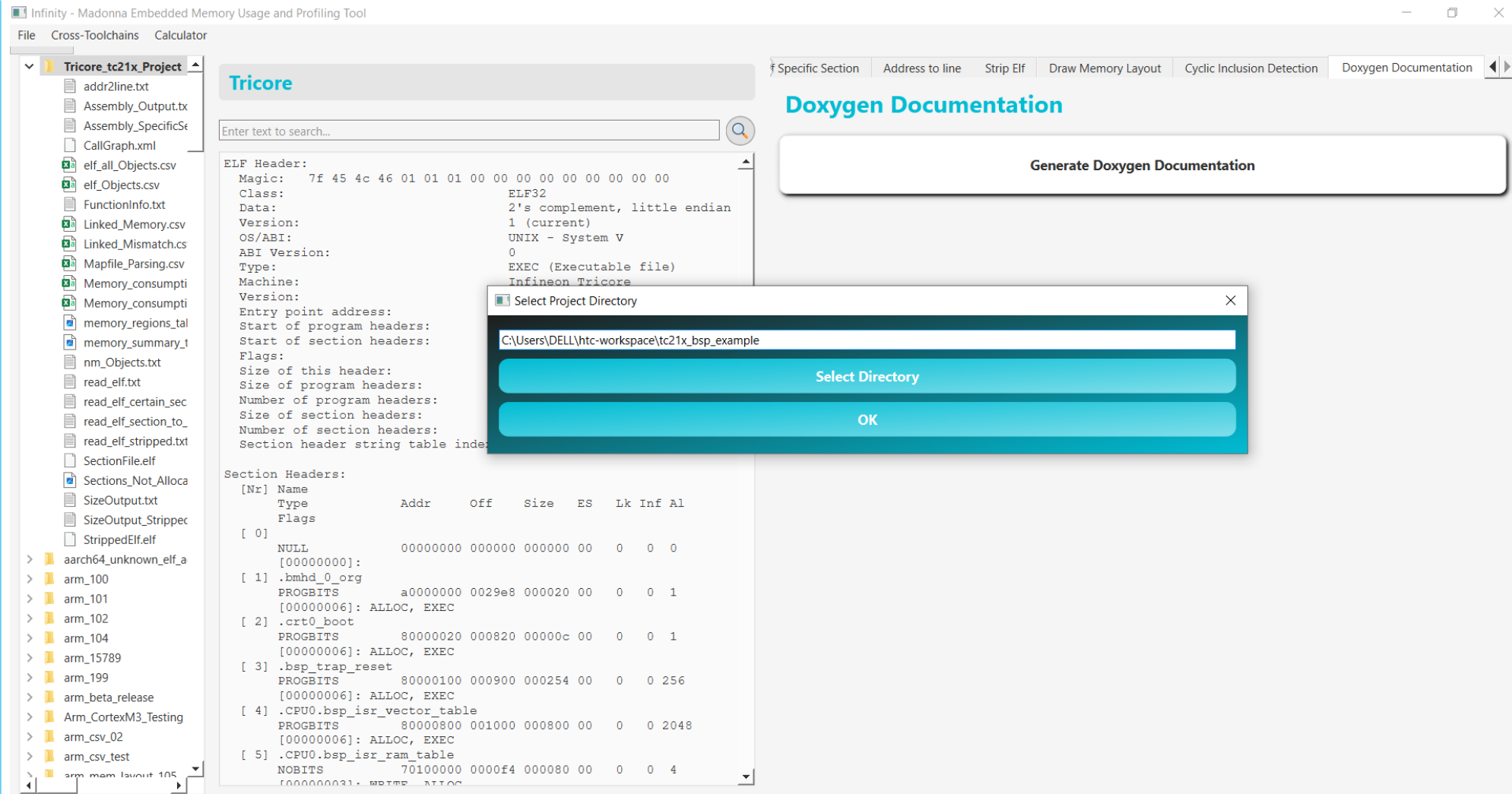
- The graph is analyzed for **circular references**, where files directly or indirectly include each other in a loop.
- The tool identifies:
 - The files involved in the cycle.
 - The specific inclusion paths that create the loop.

Cyclic Inclusion Detected - Cycle 1



∞ Generating Doxygen Documentation ∞

The Doxygen Documentation feature generates automated, Doxygen-compatible documentation from source code, improving code understanding by explaining functions, variables, and the overall structure. This makes it easier for developers to explore the codebase, fostering team collaboration, knowledge sharing, and new developer onboarding. Additionally, the Call Graphs feature visualizes function call relationships, helping developers analyze control flow, identify redundant calls, and optimize performance. Together, these features enhance code documentation, improve design and performance, and promote best practices in memory management and coding efficiency.



∞ Call Graphs – Doxygen ∞

My Project: C:/Users/DELL/htc-work X +

file:///D:/Infinity_Work_Dir/TI_CGT_2000_TestOverflow/Doxyfile_output/html/shared_main_8c.html

My Project

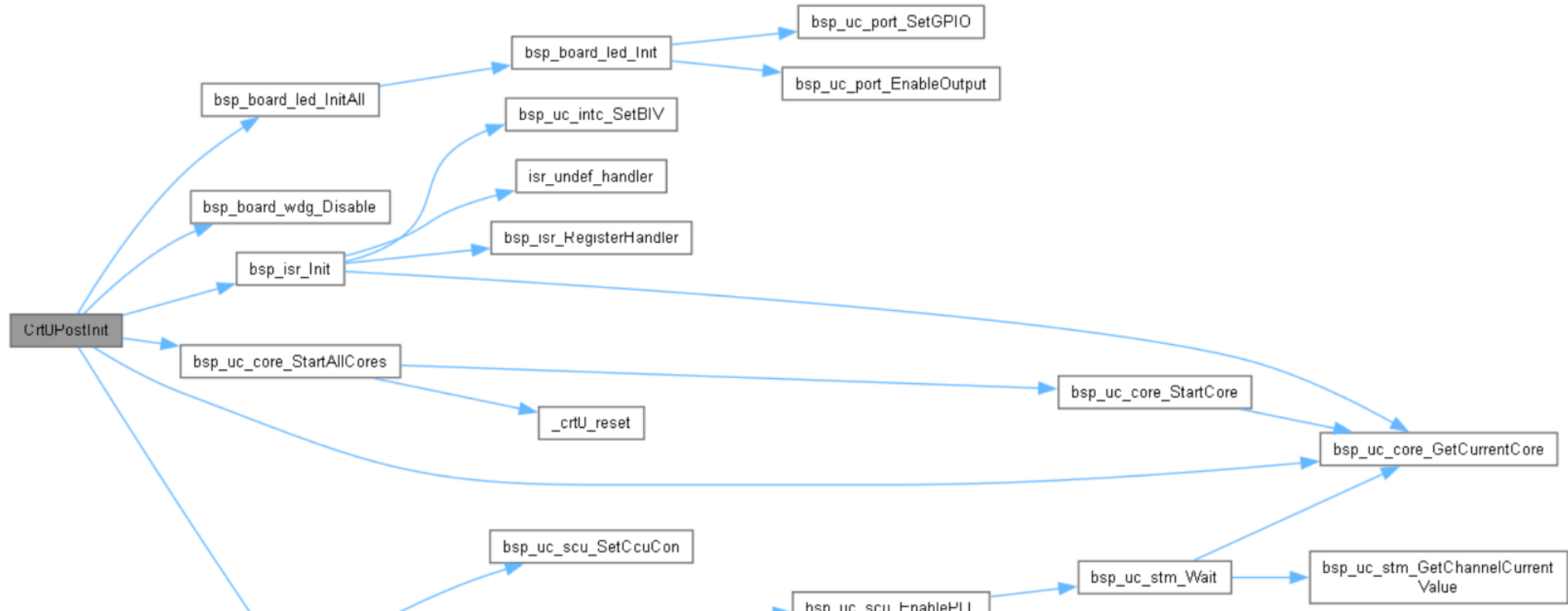
Main Page Topics Classes Files Search

C: > Users > DELL > htc-workspace > tc21x_bsp_example > src >

◆ Crt0PostInit()

void Crt0PostInit (void)

Here is the call graph for this function:



∞ Infinity is also a learning tool ∞

nm Utility and Linker Explanations

Understanding the nm Utility and Its Linker Interaction

Purpose and Use of nm

The main purposes of nm are:

- Pre-Link Inspection:** Inspect object files (.o files) to identify symbols defined within them and any external symbols they reference before linking.
- Post-Link Verification:** Check the final binary (e.g., .out or .elf files) to show resolved symbols, their addresses, and sizes.
- Debugging Missing Symbols:** Locate undefined or missing symbols and troubleshoot linker errors, such as "undefined reference."

How nm Relates to the Linker

- Before Linker (Compilation Stage):**

Object files generated from source code (like main.o and utils.o) contain machine code and a symbol table. The nm tool can inspect these object files to display symbols, including function names, variables, and section details, identifying missing symbols.
- During Linker:**

The linker combines object files and resolves symbol references between them. It locates undefined symbols across files and libraries:

 - If a symbol is found, the linker assigns it an address, includes it in the binary, and updates the symbol table.
 - If a symbol is missing, the linker produces an "undefined reference" error.
- After Linker (Post-Link Stage):**

After linking, the final binary contains a symbol table with all resolved symbols and addresses. Running nm on this binary verifies symbol resolutions and addresses.

Handling Symbols That Don't Appear in the Symbol Table

If certain object files or symbols are not linked into the final binary:

- Static Linking:** Only symbols from explicitly passed object files or libraries are included. Unused symbols are discarded unless preserved.
- Dynamic Linking:** Symbols from shared libraries (e.g., .so or .dll files) may not appear in the symbol table unless required at runtime.

Reasons why symbols might not appear in the final symbol table:

- Unreferenced symbols may be removed by the linker to save space.
- Object files might not have been included during linking.

Impact of Optimization on Symbol Visibility

During compilation, optimizations can affect which symbols are retained in the symbol table:

- Inlining:** When functions are inlined, their symbols may disappear because they are embedded directly into the calling functions, leaving no independent symbol entry.
- Unused Code Elimination:** Also known as "dead code elimination," removes functions or variables that are not referenced, causing them to be absent in the final symbol table.
- Symbol Stripping:** When using the -s flag or strip command, debugging symbols are removed, keeping only essential symbols in the binary.
- Reduced Visibility:** Compiler flags like -fvisibility=hidden may limit visibility of certain symbols, making them inaccessible to external inspection tools.

Close

Infinity serves as an interactive learning tool designed to help developers better understand memory profiling and management concepts. By providing detailed insights into memory usage, section allocation, and optimization strategies, it educates users on best practices for efficient memory management in embedded systems.

Show Tips and Tricks

Effect of GNU Compiler and Linker Flags on Symbol Table Generation

Compiler Flags

-ffunction-sections

- Purpose: Places each function into its own unique section in the object file.
- Effect on Symbol Table:
 - Each function is assigned to a separate section (e.g., .text.function_name).
 - Increases granularity with individual entries for each function section.
 - Enables finer control for the linker during optimization.

-fdata-sections

- Purpose: Places each data object (variables, constants) into its own unique section in the object file.
- Effect on Symbol Table:
 - Data objects are assigned to individual sections (e.g., .data.variable_name or .rodata.constant_name).
 - Symbols are tied to their specific sections for selective inclusion or exclusion by the linker.

Linker Flag

--gc-sections

- Purpose: Instructs the linker to perform garbage collection of unused sections.
- Effect on Symbol Table:
 - Removes symbols for unused sections, reducing the size of the final symbol table.

Close