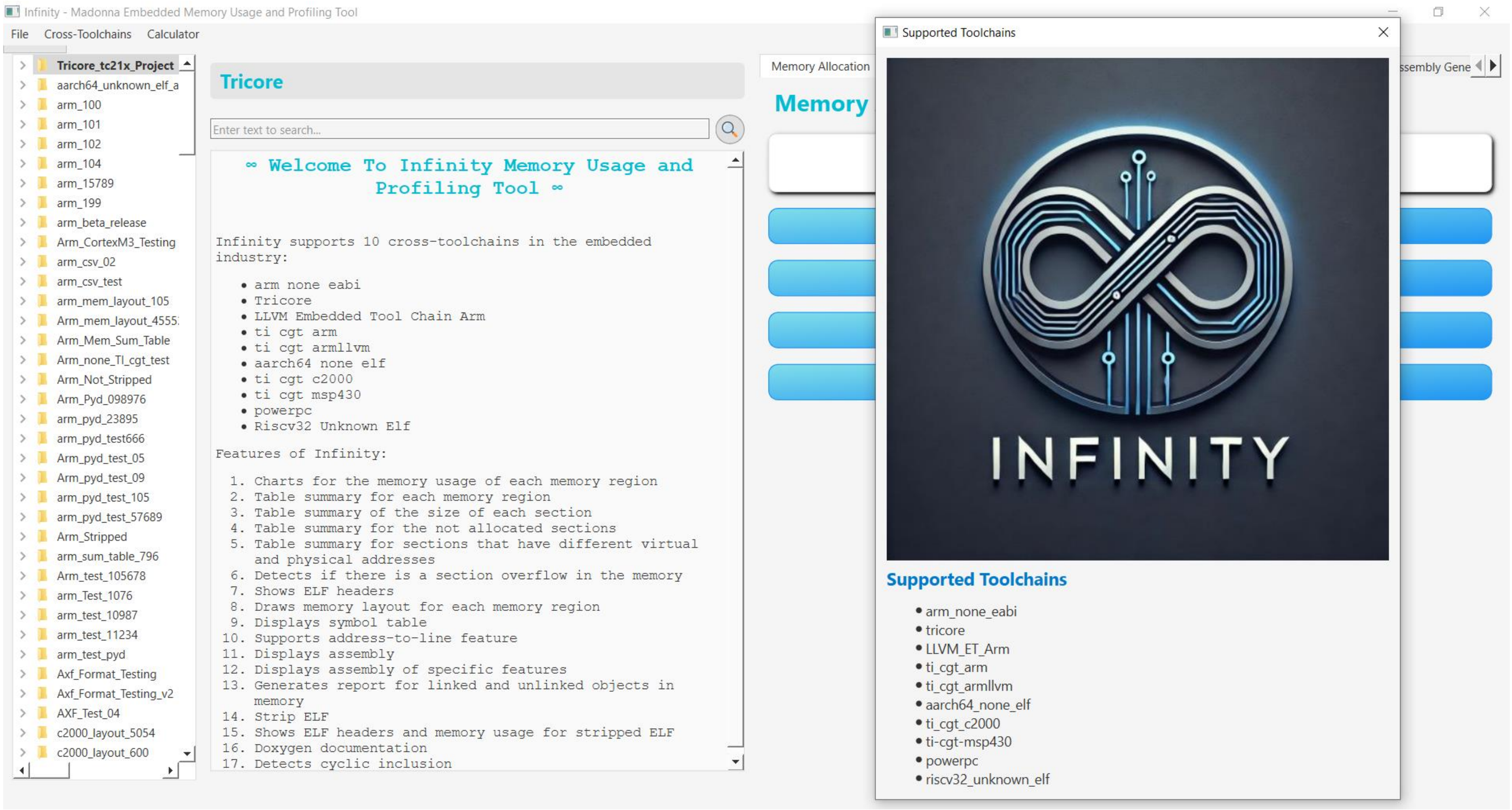
∞ Infinity- Embedded Memory Usage and Profiling Tool that Supports 10 Cross-Tool Chains ∞



∞ 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.

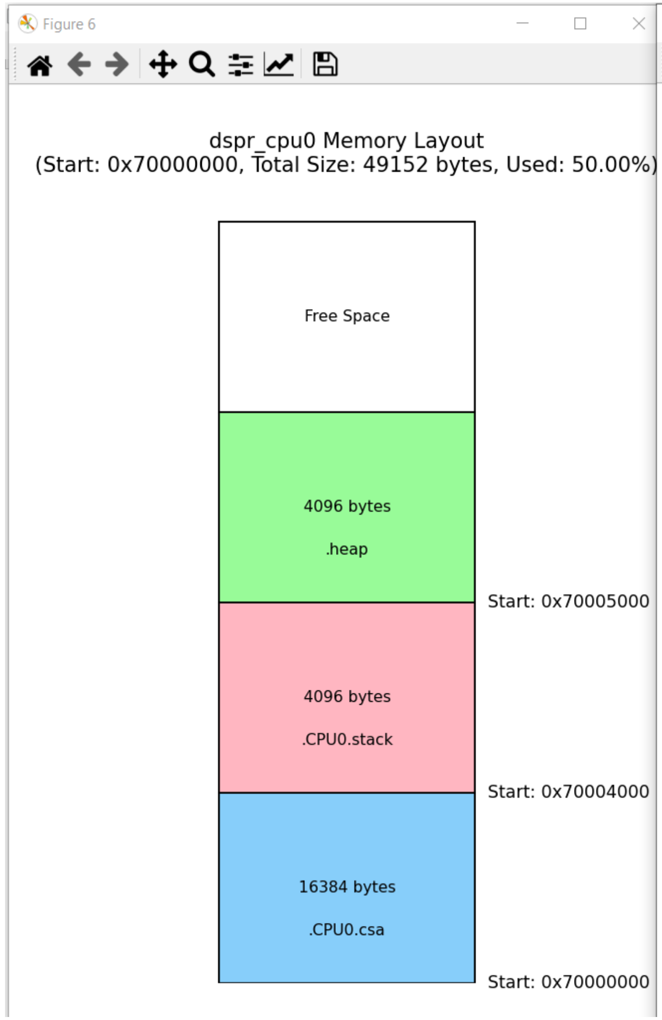
Infinity Features List:

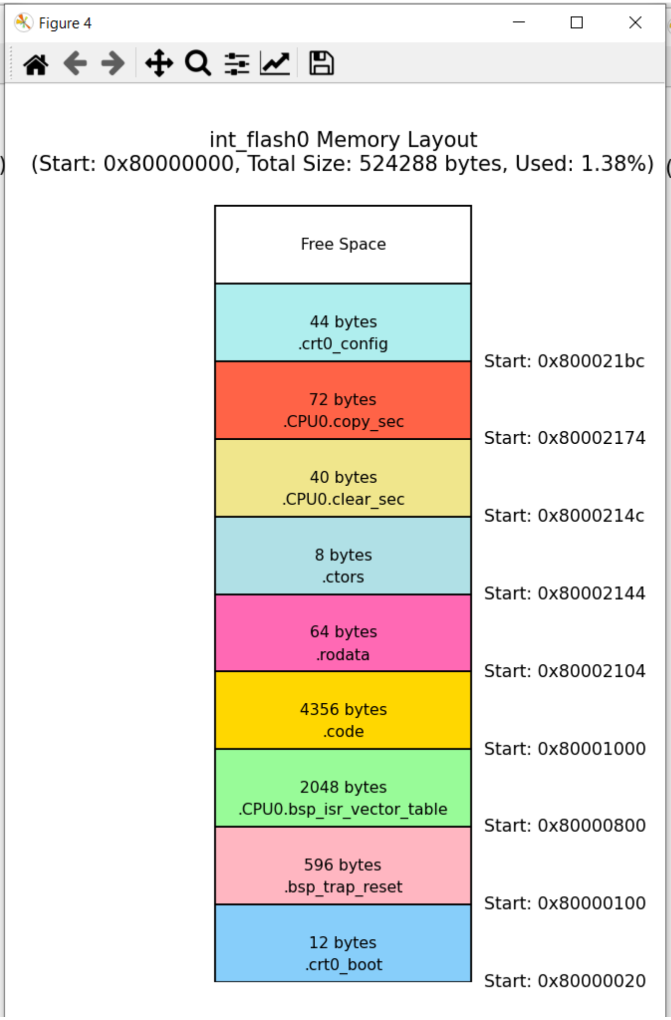
1. **Memory Layout Visualization**
2. **Comprehensive Memory Usage Analysis**
3. **Section Overflow Detection**
4. **Memory Usage Table** (Displays memory consumption for each section and region)
5. **Section Usage Table** (Shows detailed memory usage statistics for each section, including free space 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)

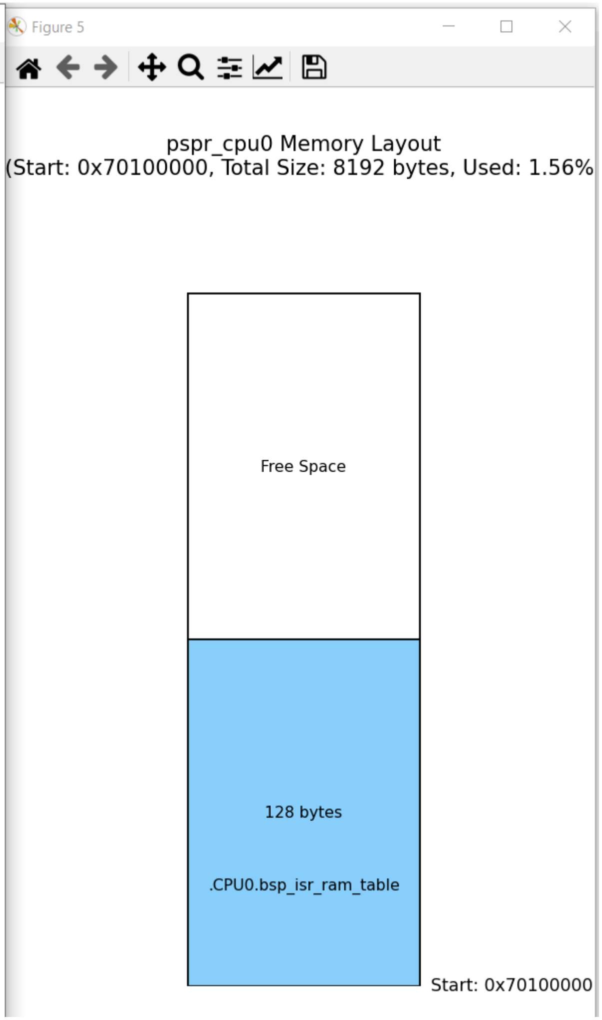
∞ Drawing Memory Layout Feature ∞

Simplifies Understanding of Memory Allocation:
Developers can easily understand which sections occupy which memory regions and how much space is left unused.

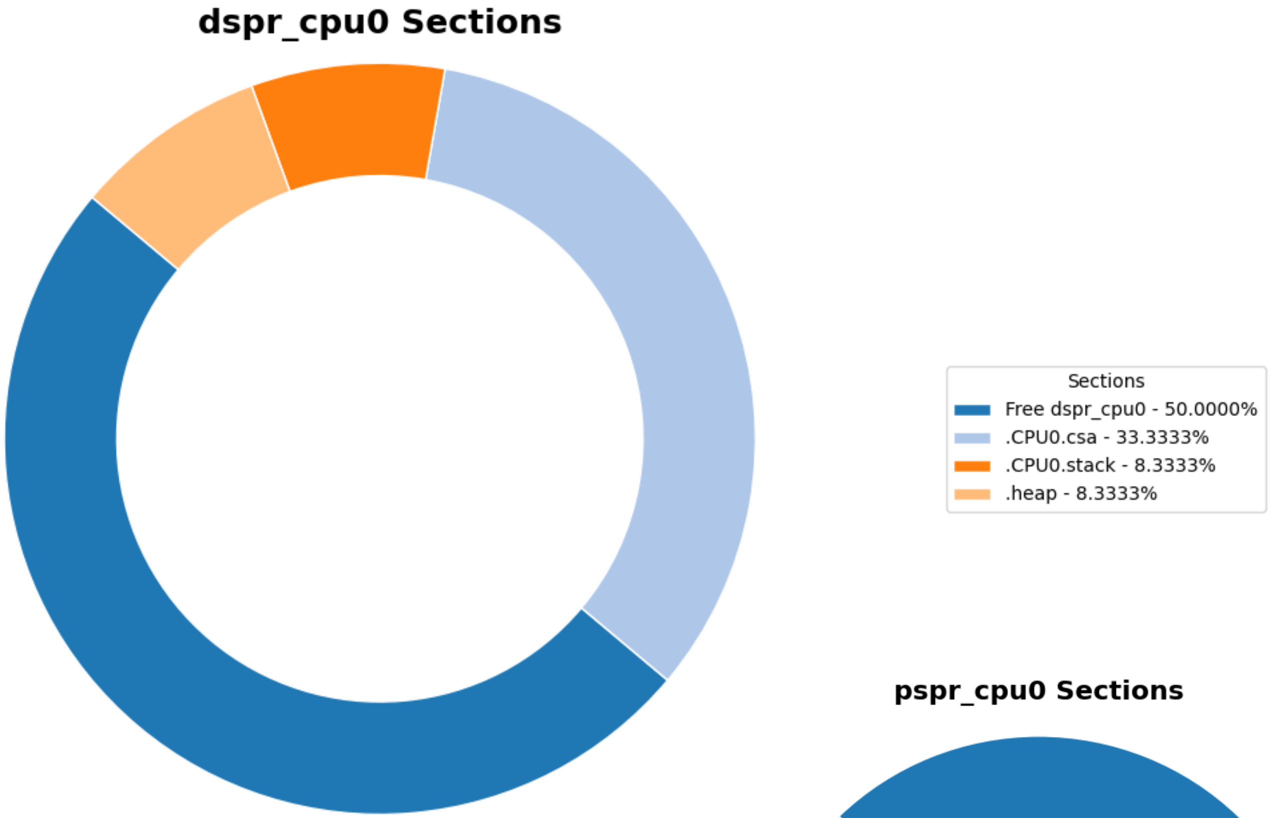
Identifies Underutilized or Overused Regions Quickly:
Overused memory regions can cause overflow, leading to instability or crashes, while underutilized regions indicate inefficient resource usage, potentially increasing costs. This feature helps developers quickly identify and address these issues by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

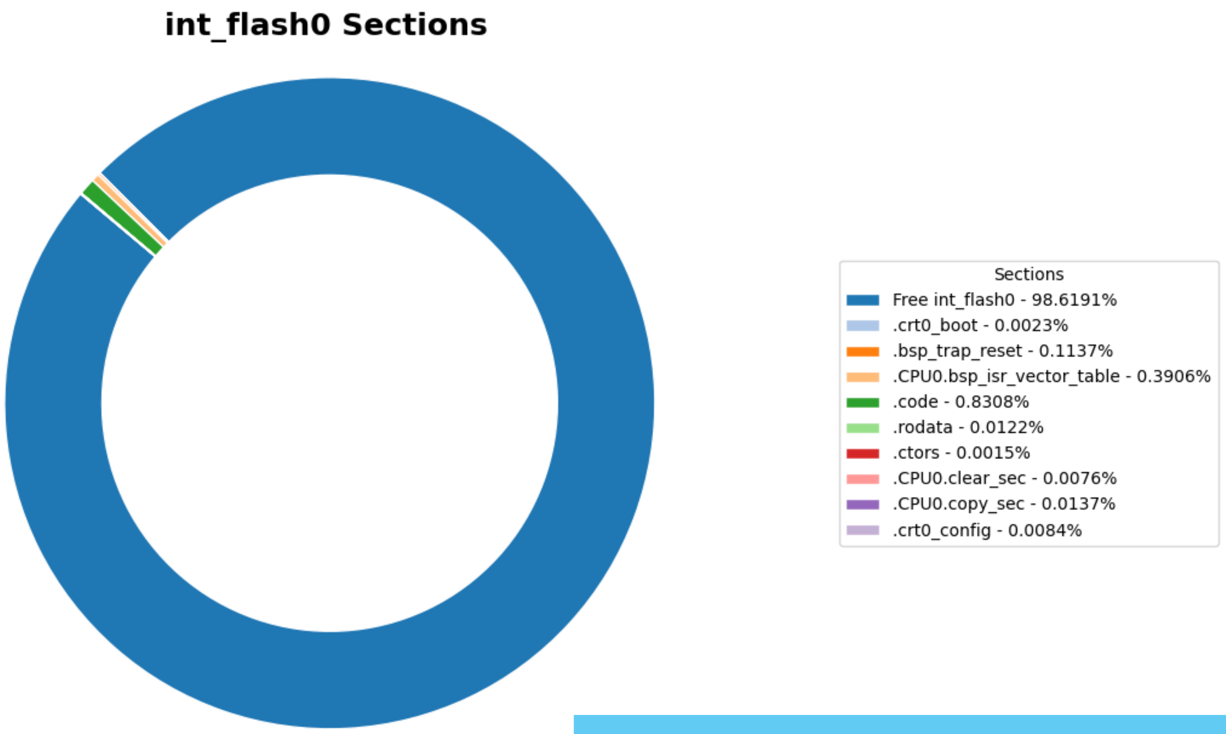


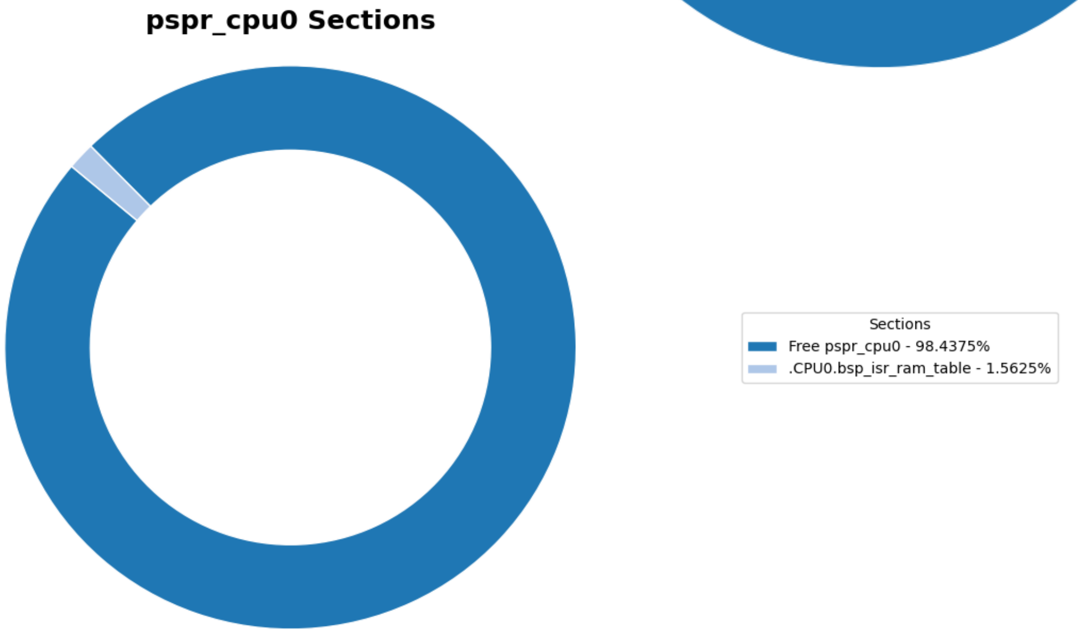




∞ Memory Usage Charts ∞







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 ∞

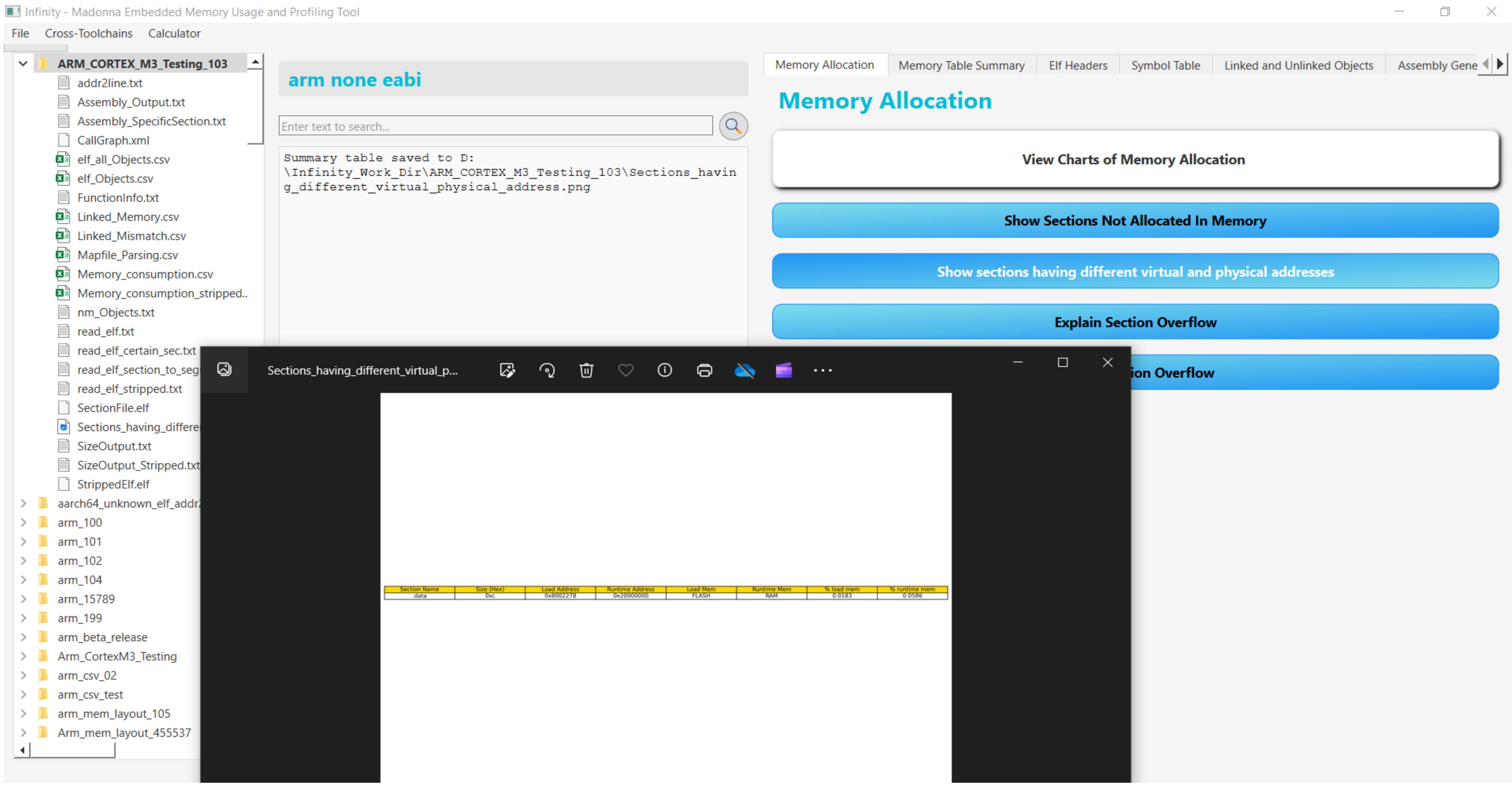
This feature identifies and reports sections where the **virtual address (VA)** and **physical address (PA)** differ. It provides a detailed mapping of these addresses for each section.

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.

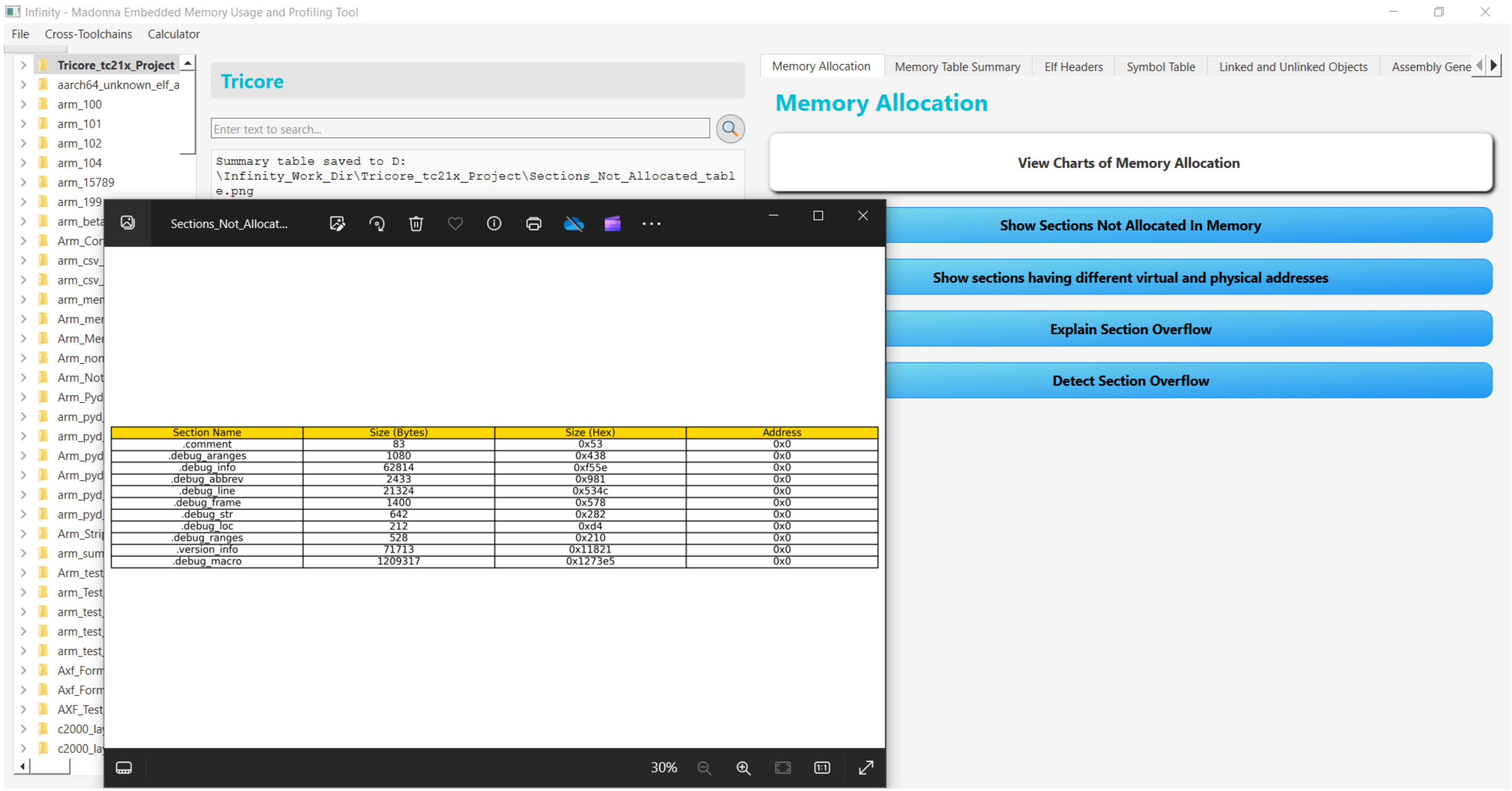


Section Name	Size (Hex)	Load Address	Runtime Address	Load Mem	Runtime Mem	% load mem	% runtime mem
.data	0xc	0x8002278	0x20000000	FLASH	RAM	0.0183	0.0586

∞ 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.



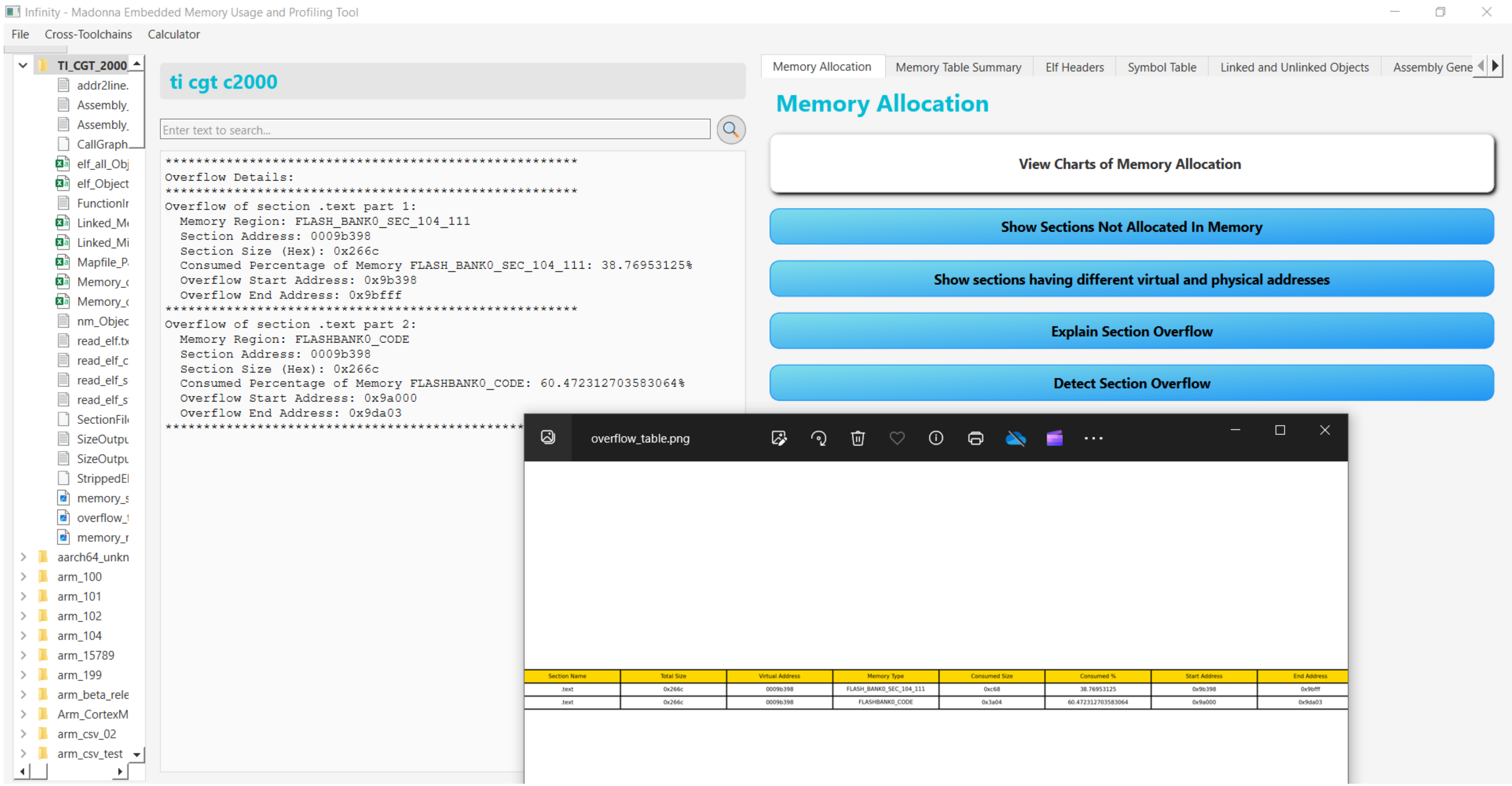
Section Name	Size (Bytes)	Size (Hex)	Address
.comment	83	0x53	0x0
.debug_aranges	1080	0x438	0x0
.debug_info	62814	0xf55e	0x0
.debug_abbrev	2433	0x981	0x0
.debug_line	21324	0x534c	0x0
.debug_frame	1400	0x578	0x0
.debug_str	642	0x282	0x0
.debug_loc	212	0xd4	0x0
.debug_ranges	528	0x210	0x0
.version_info	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.

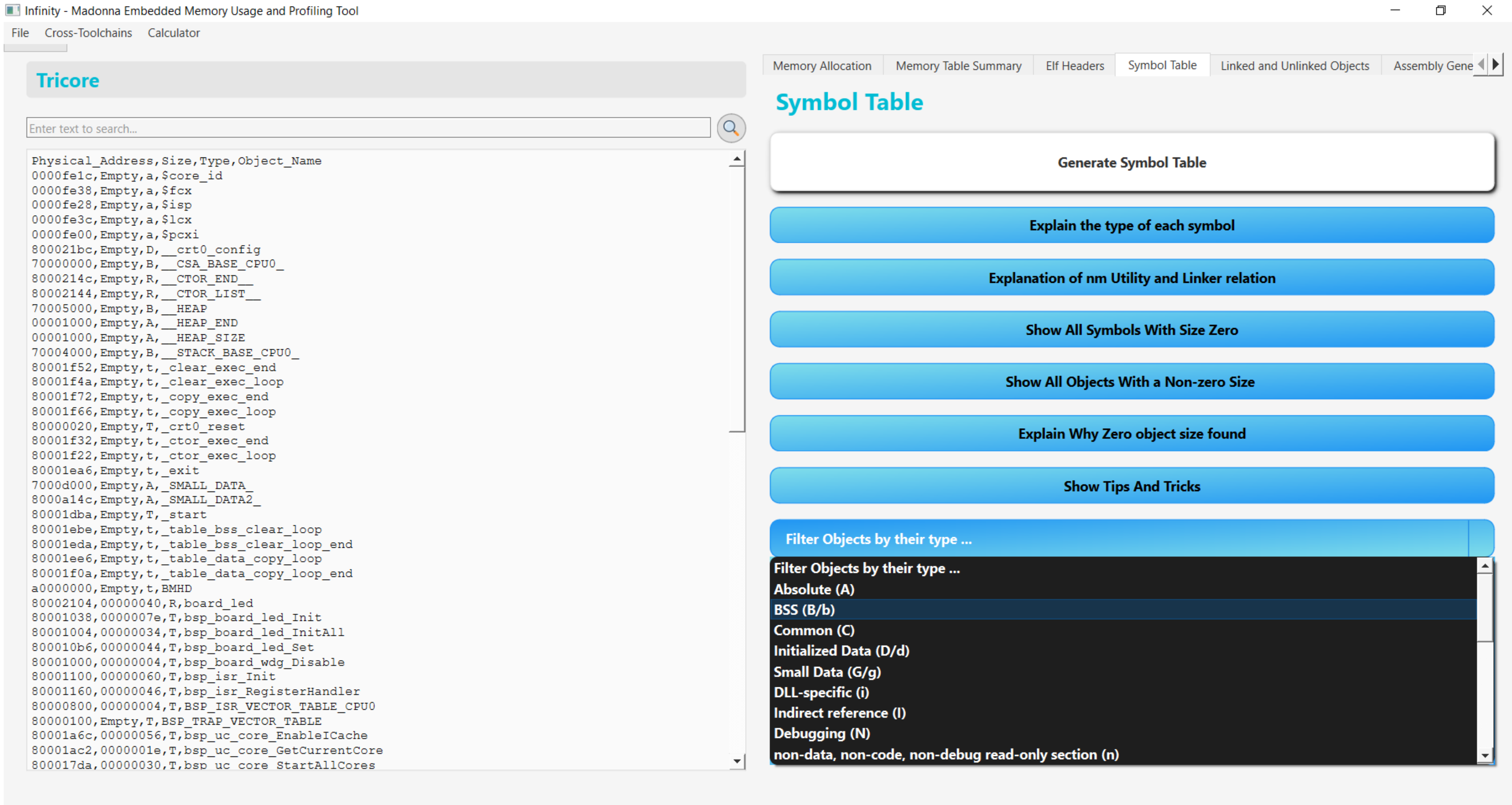


∞ Symbol Table With Filtration of each Symbol Type ∞

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.



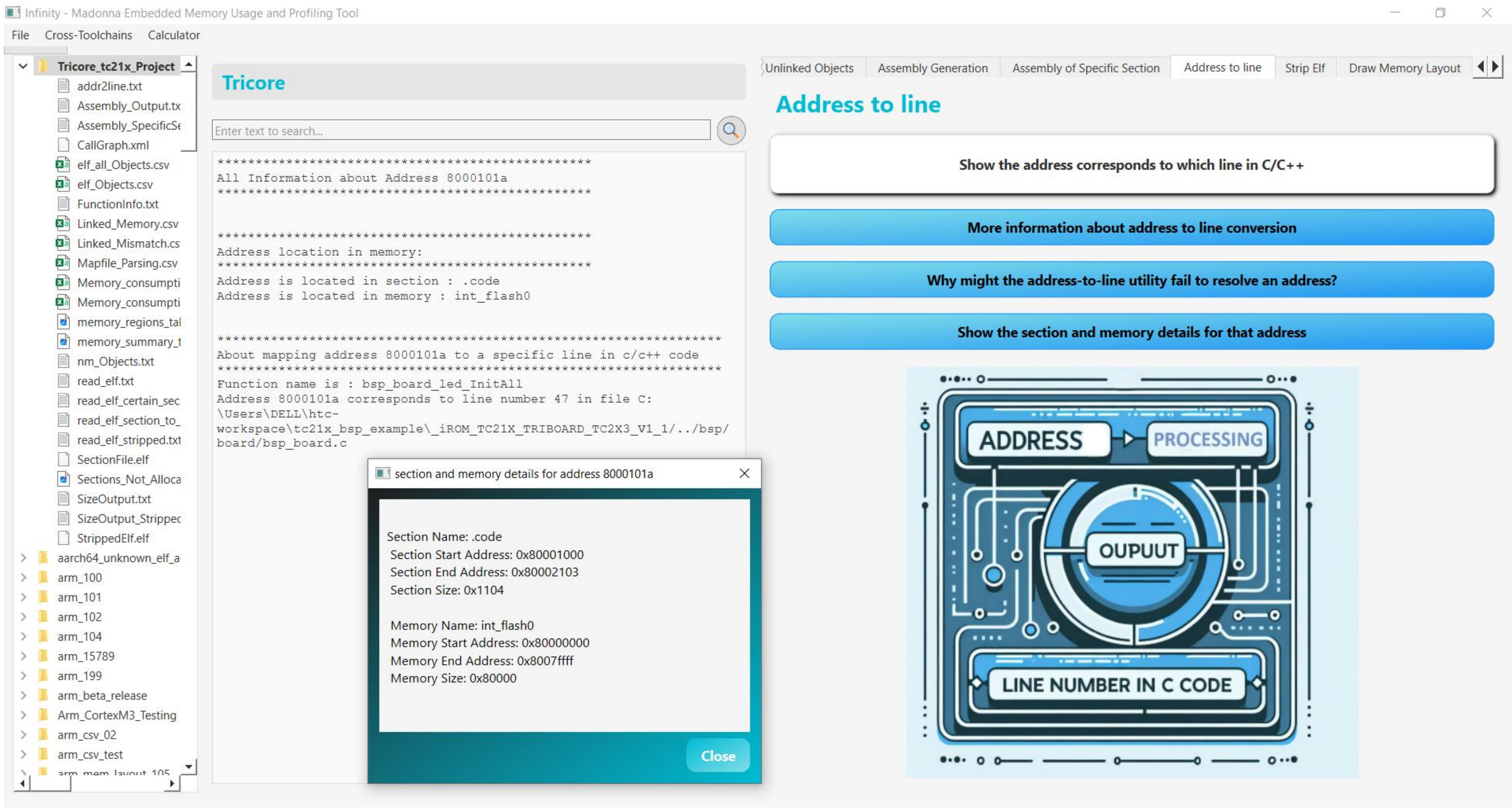
∞ Mapping the desired address to a specific line in c code ∞

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:

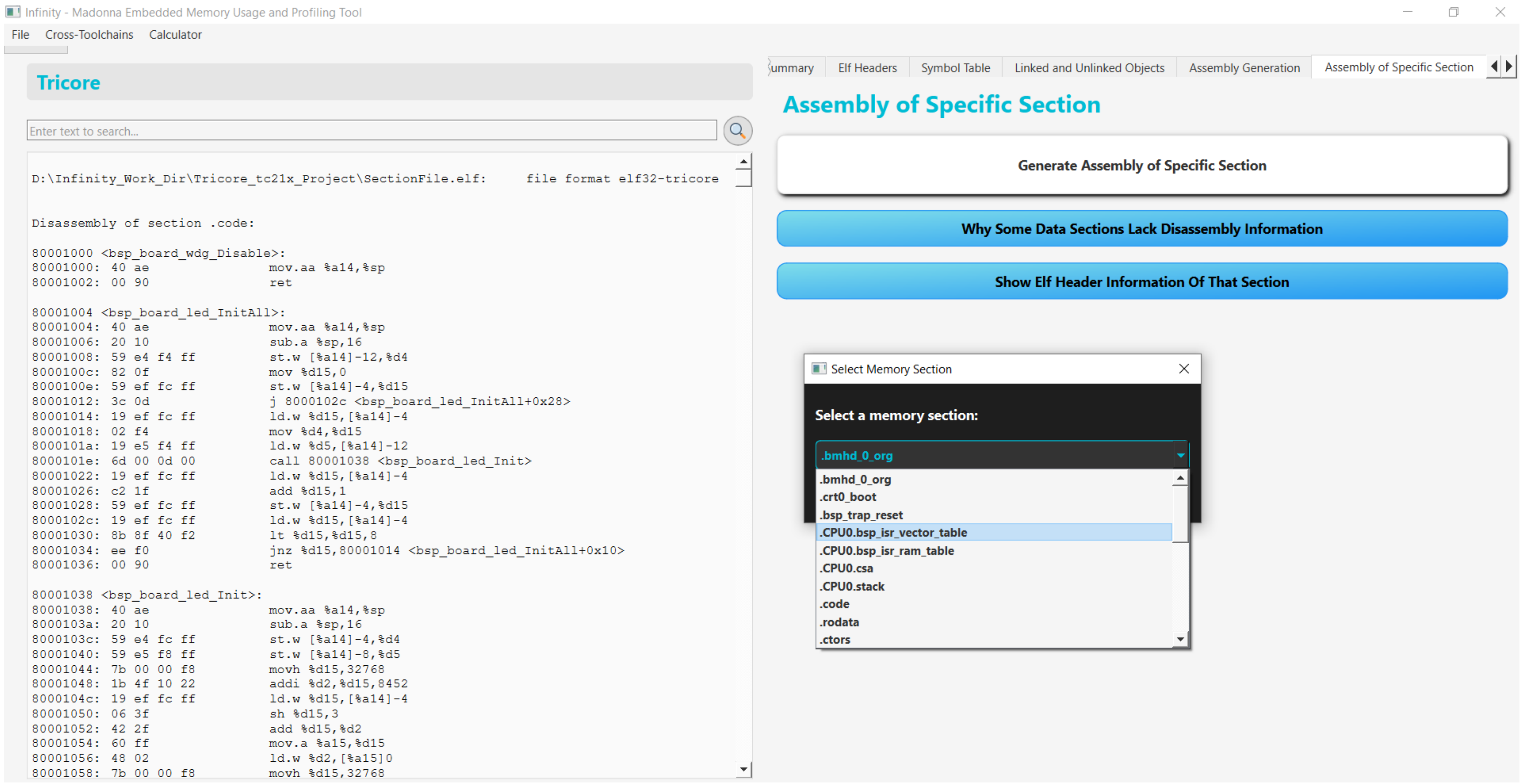
- **Address-to-Code Mapping**: The tool identifies the exact line of C source code corresponding to a given memory address.
- **Section and Memory Information:** Alongside the source code mapping, the tool also provides: The memory section (e.g., .text, .data, .bss, .rodata) where the address resides. Additional details such as the start address, size, and attributes (e.g., writable, executable) of the section containing the specified address.



∞ 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.

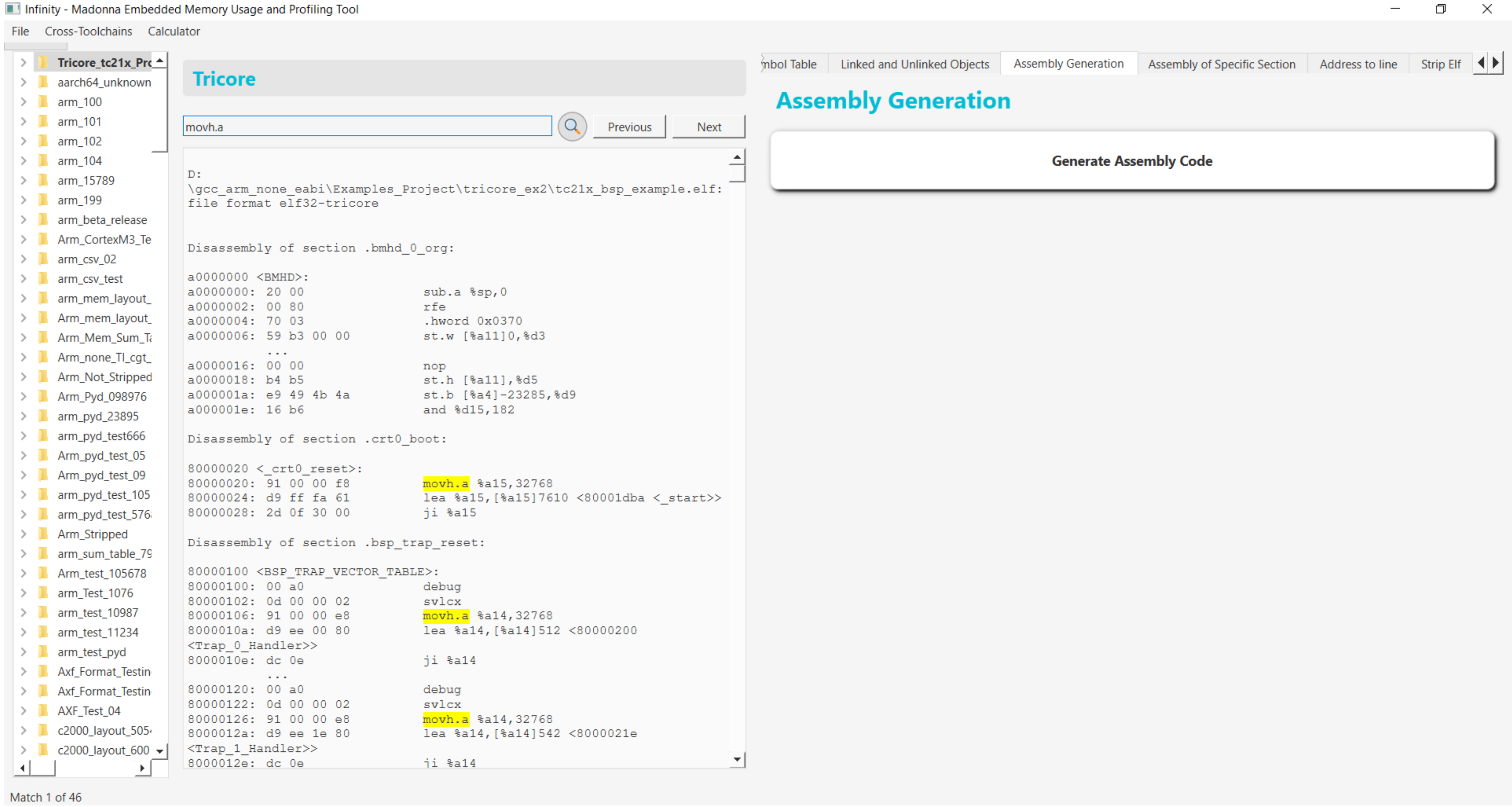
Along with the assembly, **the tool provides relevant ELF header information for the selected section**, such as its address, size, alignment, and attributes, ensuring a comprehensive understanding of the section's role in the binary.



∞ 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.



∞ 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:

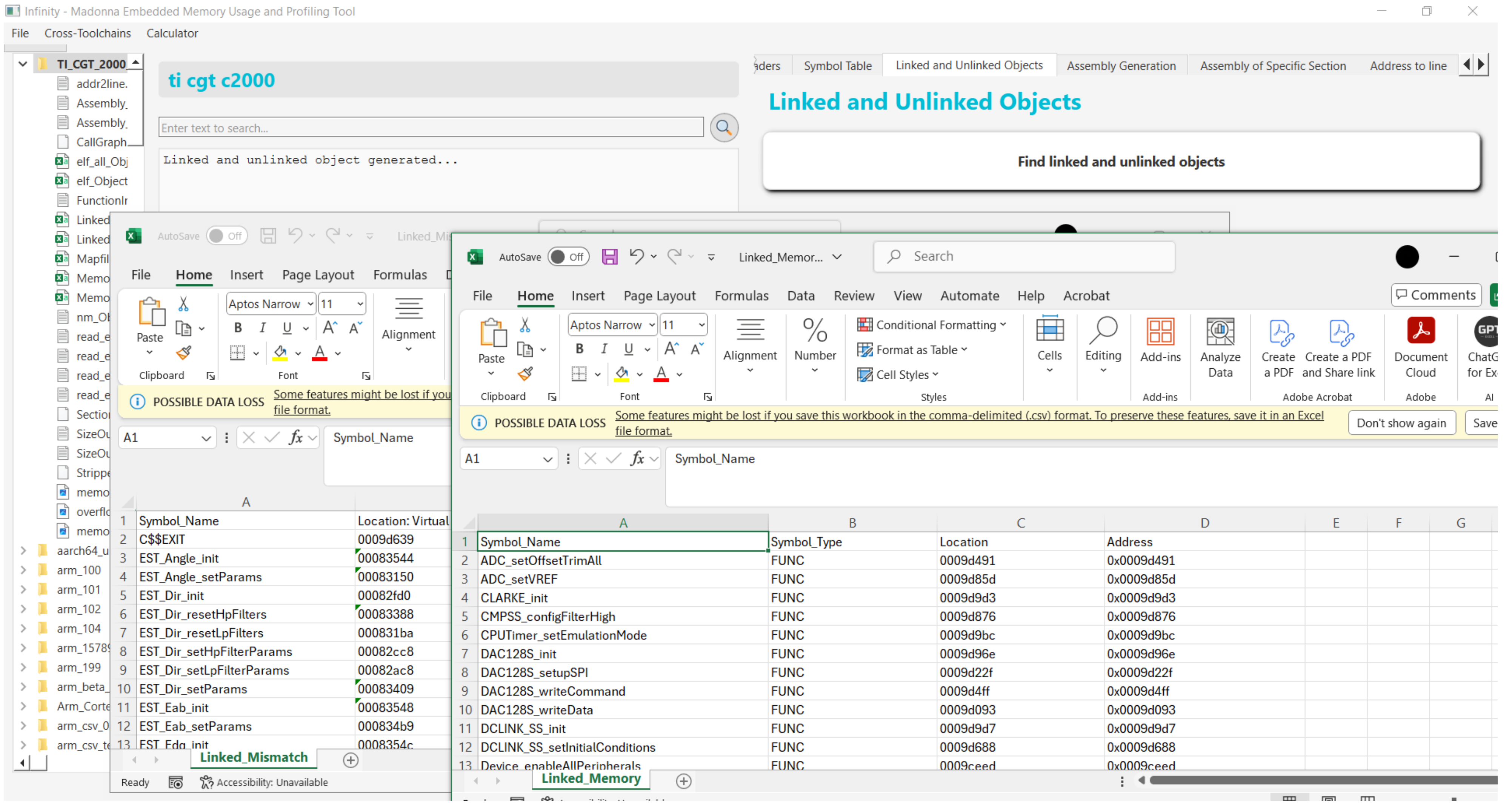
Unused Code: Functions or variables that are defined but never referenced.

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

By documenting unlinked objects, developers can:

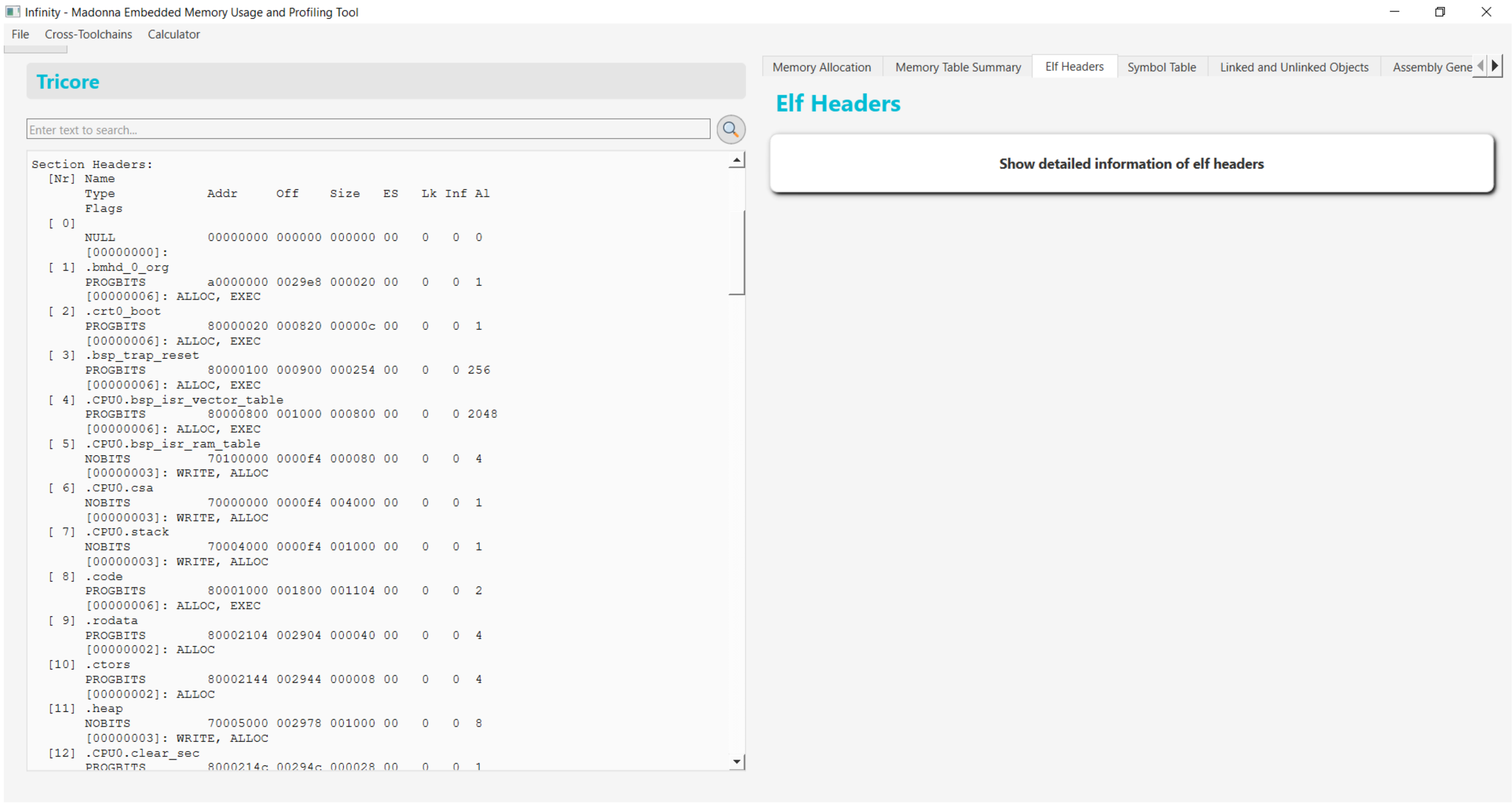
Remove unnecessary objects to clean up the codebase.

Ensure that no critical objects are accidentally excluded.



∞ 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.



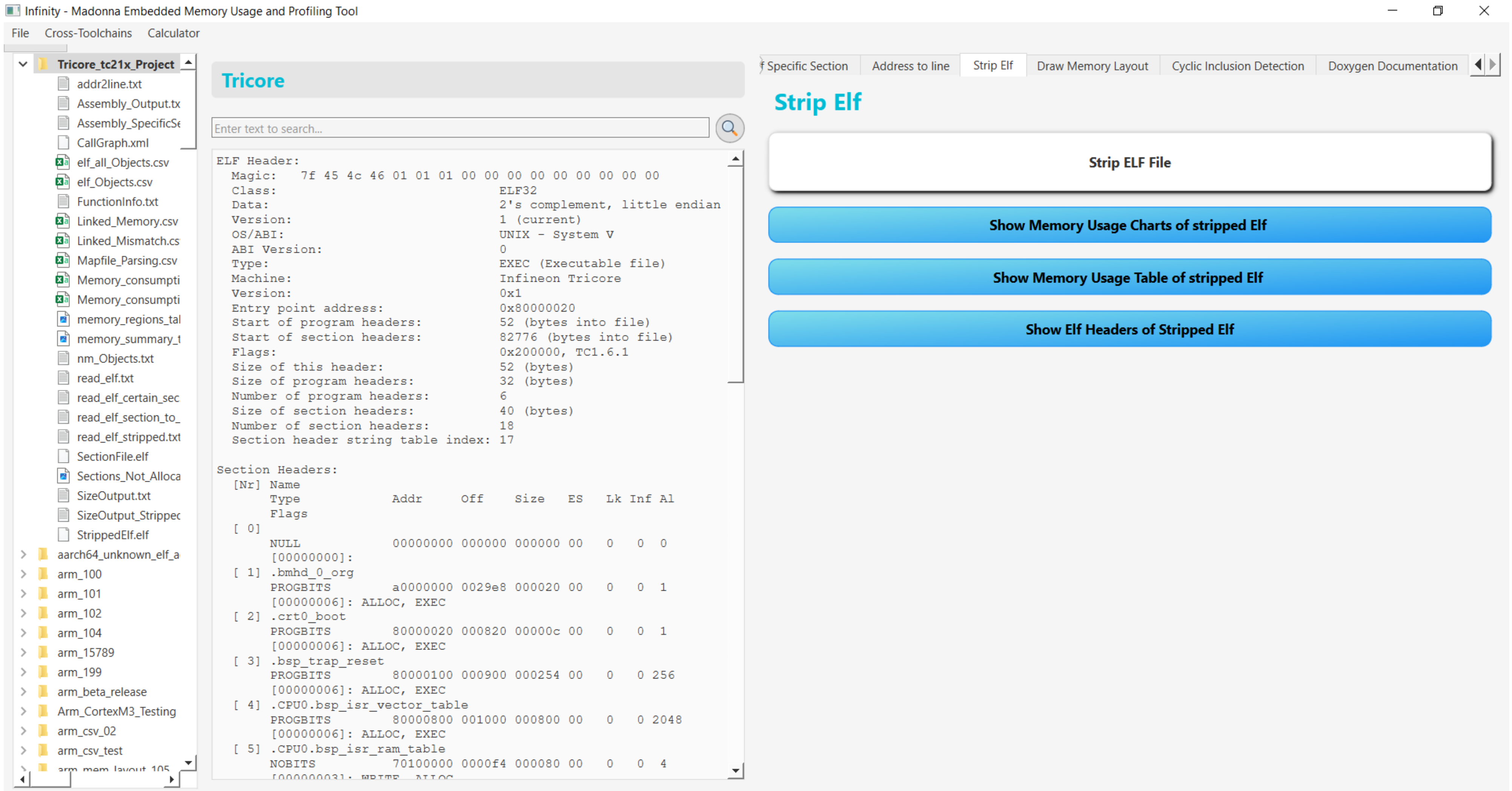
∞ Stripping Elf ∞

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.

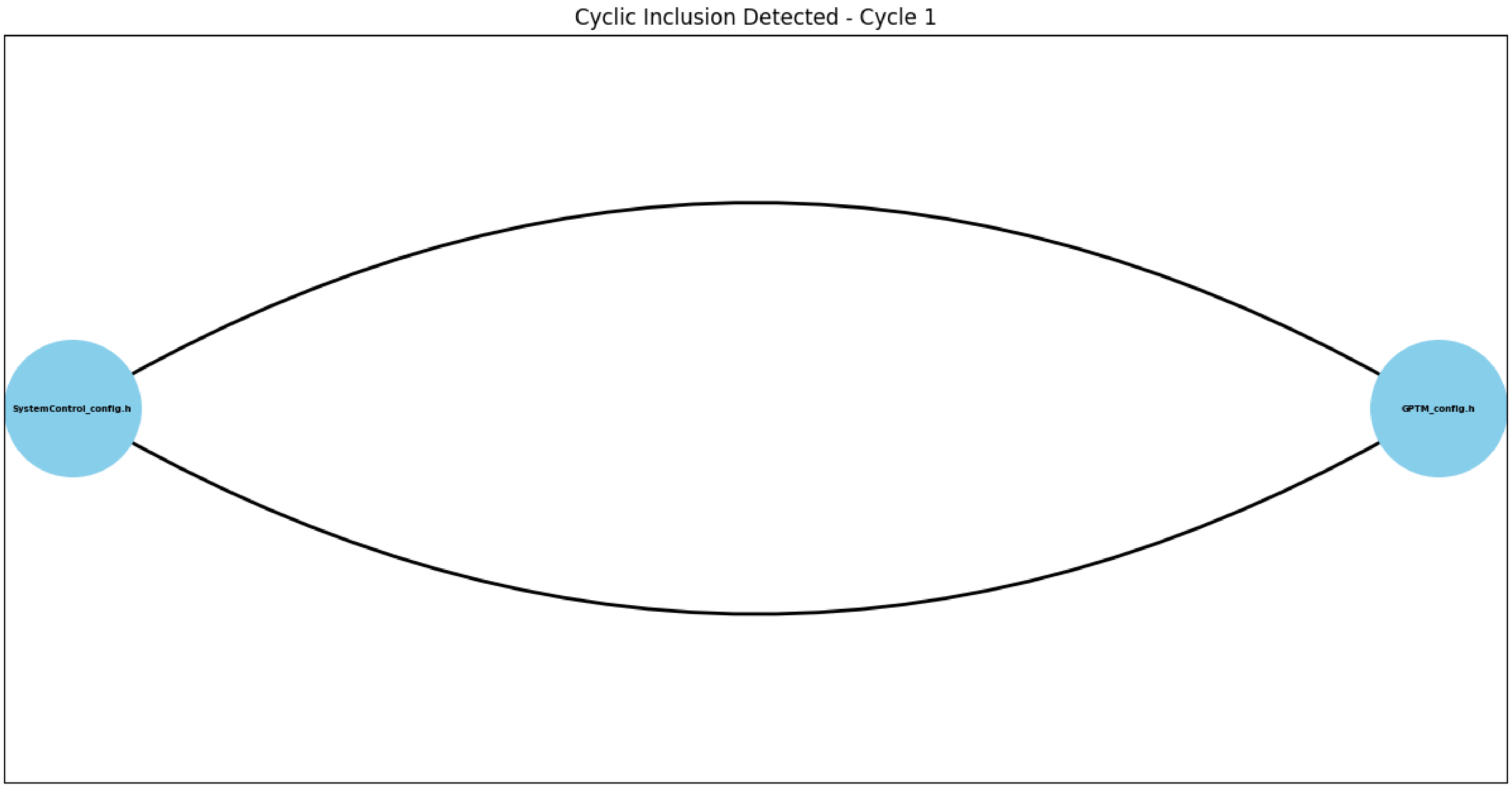
•Stripping removes this information, reducing the risk of reverse engineering.



∞ Detecting Cyclic Inclusion ∞

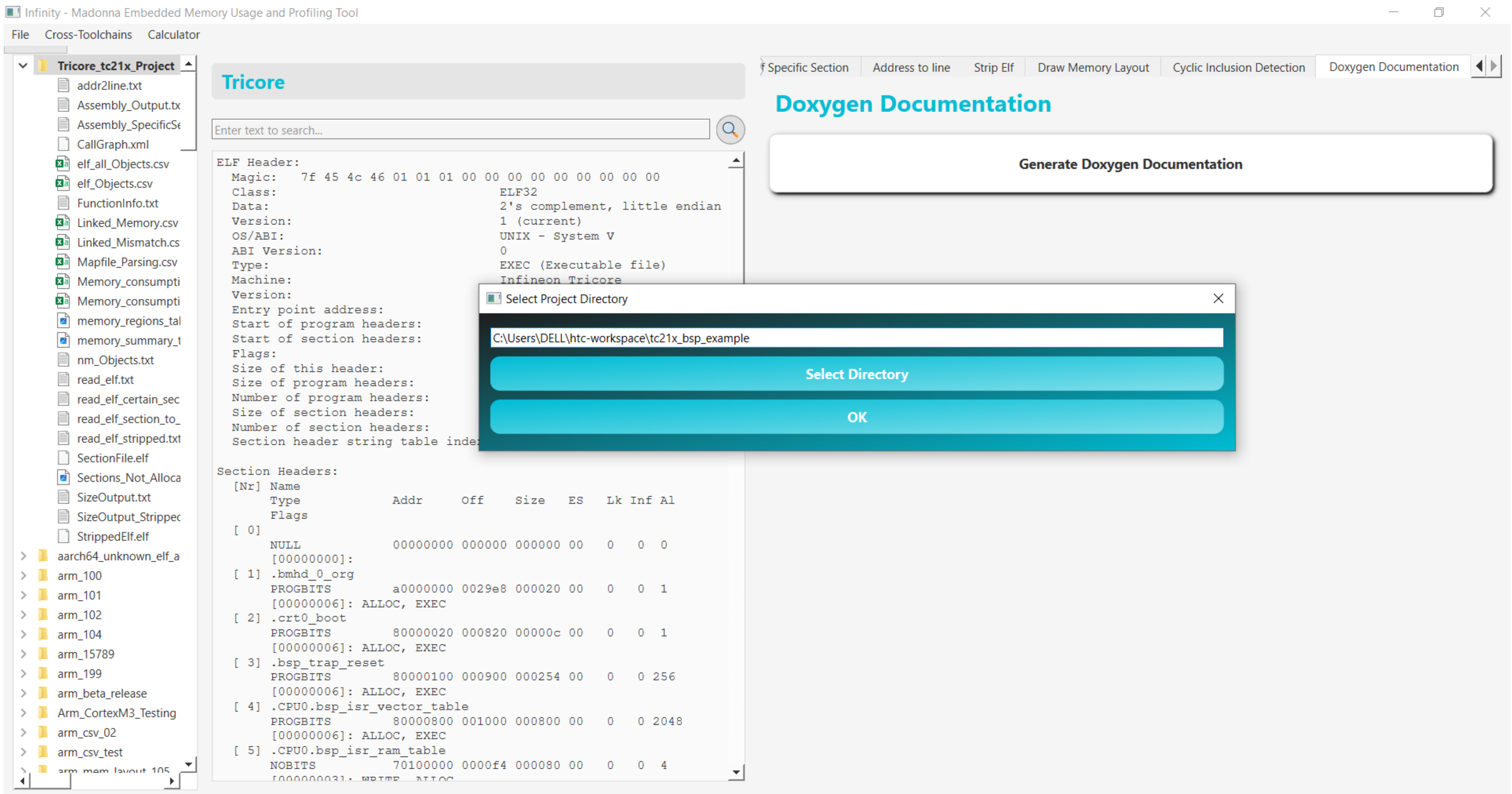
The **Detecting Cyclic Inclusion** feature identifies and resolves **circular dependencies** in header files, modules, or source files. Cyclic inclusion occurs when two or more files reference each other directly or indirectly in a loop, leading to an **infinite inclusion cycle** during the preprocessing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

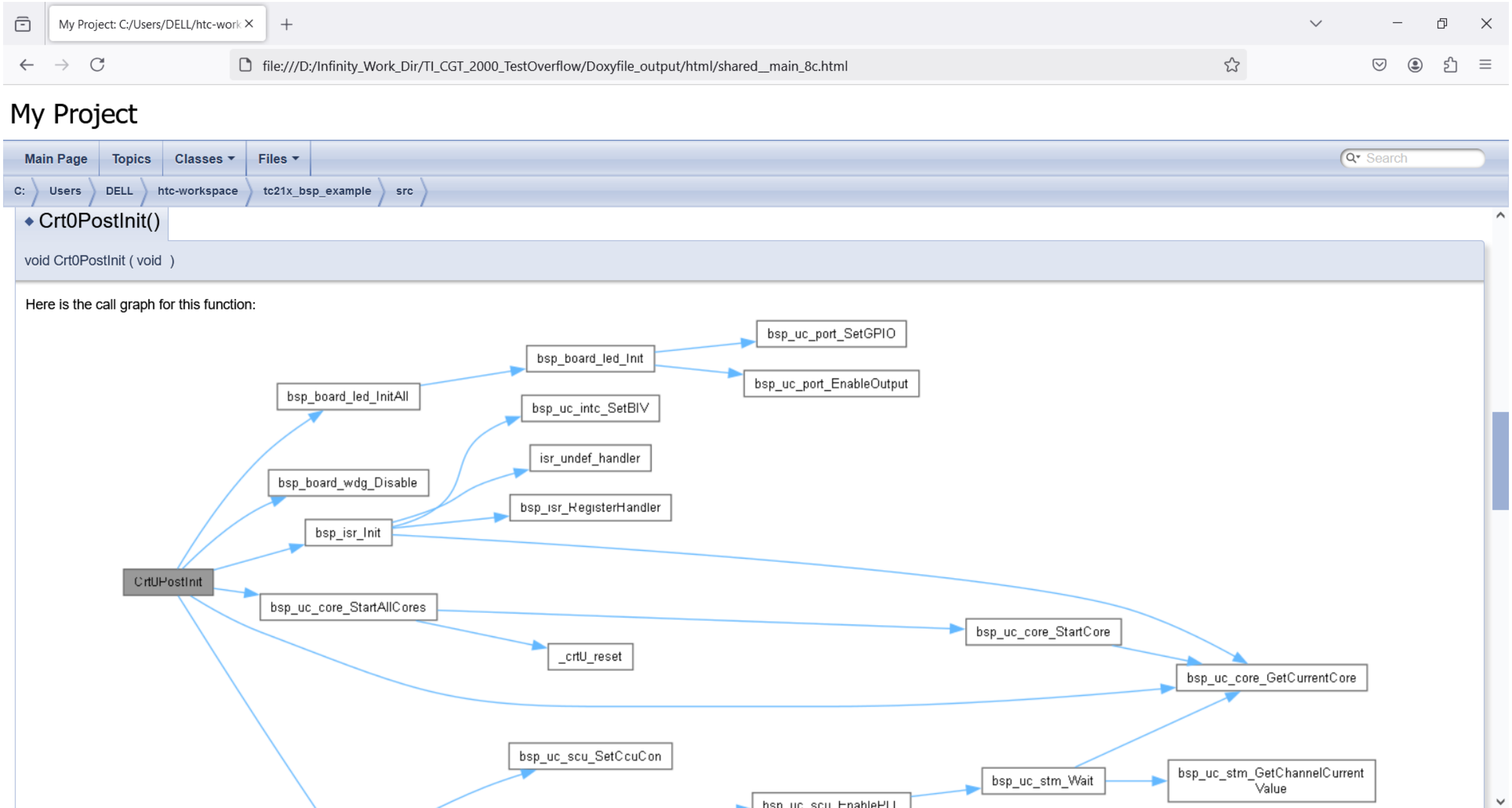


∞ Generating Doxygen Documentation ∞

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 ∞



∞ 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:

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

How nm Relates to the Linker

1. 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 symb

2. 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.

3. 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