Debugging NMF Applications with TRACE32

Debugging NMF Applications with TRACE32

Version 18-Nov-2020

This document describes the support provided by TRACE32 for debugging applications using the NMF (Nomadik Multimedia Framework). It only handles the aspects of debugging on MMDSP. A general understanding of system level topics like the interaction with the ARM core should be given before reading it.

The main feature of the NMF framework is to load and unload components at run-time onto the MMDSP core. The components will be loaded onto different addresses depending on the available memory ressources. Due to this, program and data symbols need to be relocated by the debugger i.e. the debug info of the .elf file (starting always from 0x0) is internally adapted to the base address which the symbols are loaded at.

For details on the data structures please refer to the documentation provided by STM. This application note explains the commands and features of TRACE32 pertinent to debugging of NMF applications.

Terminology and Syntax

Programs

In the jargon of TRACE32 we speak of programs, that correspond to .elf files. A program corresponds to a NMF component. A program can have various modules, that correspond to linkage units i.e. object files and libraries that are linked together to form the executable .elf file. The hierarchy of these modules is preserved in the naming scheme for symbols.

Components

In NMF jargon we speak of loading and unloading "components" at run-time. Such a component is represented by an .elf file, normally also including debug information. The corresponding term in TRACE32 is that of "program". Both terms are used interchangeably throughout this document.

In NMF a component can be instantiated multiple times, each time with a different data section. In this document such components will be referred to as **component instances** or simply as **instances**.

The name of an NMF component is composed of its .elf-file name, plus its file path i.e. the names of the directories under which it is located (like in JAVA or C#). This naming scheme was chosen in order to distinguish between components with identical names. However, the debugger does not use the raw names, but does some "beautification" (i.e. exchanging slashes by `$`, removing `.elf` appendix) to make it compatible with TRACE32 syntax conventions. For example for referring to the .elf file.

```
dec_one/adder/adder.elf
```

in TRACE32, the following name is used

```
dec_one$adder$adder
```

Note that this component name **by convention** is also used to name the corresponding program (via **Data.LOAD.Elf ... /NAME** *<program_name>*) such that “component name” and “program name” can be used interchangeably.

Naming Syntax in TRACE32

The general syntax for specifying symbols (variables, functions, ...) in TRACE32 is as follows:

```
\\ProgramName\ModuleName\SymbolName
```

Samples for global symbol names are:

```
\\adder\quickadder\m
```
... a symbol m (e.g. a variable) in module quick adder of the program adder

```
\\dec_one$adder$adder\dec_one\value
```

In the context of NMF, due to the equivalence of components and program, this is identical to

```
\\ComponentName\ModuleName\SymbolName
```

Due to this equivalence, under NMF the symbols are loaded from their respective .elf files using the parameter "/name ComponentName" thus naming the loaded program according to the component name.

```
Data.LOAD.Elf adder.elf  /NAME "dec_one$adder$adder" /NoCODE
```

NMF Execution Engine (EE)

The NMF execution engine is a small module in MMDSP memory that takes commands from the ARM core (i.e. from a corresponding NMF engine) e.g. for loading or unloading of modules, updating of the loadmap etc. The NMF EE itself is implemented as an NMF module and thus contains an entry in the loadmap (with names like `synchronous_8815_hsem`). It is loaded when the first "user component" is instantiated at the DSP. Therefore, before loading the first user component, the NMF loadmap is empty.

TRACE32 Features for NMF

For debugging NMF applications, TRACE32 supports the following features:

- Autoloading and unloading of symbols for dynamically loaded components
- Managing of loaded components in the **sYmbol.AutoLOAD.List** window
- Handling of components with multiple instances
- Inspection of variables in arbitrary, also inactive components
- Component-specific breaks and deferred breaks

These features are implemented in the TRACE32 core software, but also use supporting PRACTICE scripts (*.cmm). The advantage of this is increased flexibility and maximum control of the user over the loading process.

Autoloading and Unloading of Component Symbols

For debugging dynamically loaded modules, the corresponding debug symbols must be available. TRACE32 provides an autoloading feature that inspects the NMF load map structures in DSP-memory that indicate the currently loaded modules.

For tracking changes in the loadmap TRACE32 sets breakpoints on the NMF functions for loading and unloading components i.e. on the symbols `Constructer_Run` and `Destroyer_Run` (which belong to the NMF execution engine running on MMDSP). When loading or removing a component, the DSP will hit these breakpoints. This triggers execution of the script `updatemodules.cmm` which in turn calls **sYmbol.AutoLOAD.CHECK** to check for changes in the loadmap.

When the debugger detects a changed load map (in the call to **sYmbol.AutoLOAD.CHECK**), it will call `autoload.cmm` for each new component, passing parameters like component name, the pointer to the data area and program area. Inside `autoload.cmm` the program's symbols are loaded. When a component is unloaded its symbols and breakpoints are discarded by the debugger.

After updating the loadmap, the debugger calls a script (called `updatebreaks.cmm` by convention) to set all component specific breaks for all components that have now become available.

Managing of Components via sYmbol.AutoLOAD.List window

For each component found in the loadmap, the **sYmbol.AutoLOAD.List** window contains:

- The program name of the component and (if available) its instance name
- Range of program addresses
- The name and location of the corresponding .elf file for locating the debug symbols.
- The base address of the program sections
- The base address of the data sections (THIS pointer)
- The `autload.cmm` command used to load the module, including the parameters

By right-clicking on a component and clicking on “select component”, the component becomes active and the value of the THIS pseudo register is set to its THIS pointer value. When displaying under specified variables (with ambiguity to relative different instances), the displayed value is calculated using the actual value of the THIS pseudo register (which may differ from the value of THIS in `X:0x1`).

NOTE:	In the **sYmbol.AutoLOAD.List** window the start and end-addresses are listed. However, when a module was just loaded, the end address of the loaded component is not know yet (for debugger-internal reasons). It will be displayed after the next change to the loadmap.

Inspection of Variables in Arbitrary, also Inactive COMPONENTS

When resolving variable names (i.e. finding a specific variable in the debug info loaded for the various components) in the context of NMF we distinguish between two main types of variables:

- "local" variables declared inside functions
- Component global variables, that are visible to all functions of the **instance,** but are not visible by other instances of the same component or other components.

Local variables are handled as in normal programs: the debugger checks the current PC, determines the active function (or block inside a function) and from this searches until it finds the debug information for all variables (including parameters) in this scope.

Component global variables are handled somewhat differently because there can be multiple instances of a component. Therefore the PC alone is not sufficient to determine which variable is meant. The actual variable used also depends on the system-global THIS pointer at X:0x1.

Explicitly Specifying a Component Instance

When there are multiple instances of a component, the user can specify one of them using the following syntax variants:

- `\\$adder_instance_1` : using the instance name of the component. In case there are multiple instances with the **same name**, any one of them is chosen.
- `\\$4B2 or \\$0x3AF`: using the this pointer of component. This is useful in cases where the instance has no name or the names are not unique.

The prefix `\\$` is mandatory. Without it, the string is not recognized as component name. Note though that the functions `nmf.this()` and `nmf.isactive()` take *strings* as parameters such that the instance specifier needs to be inclosed in double quotes e.g.

```
print nmf.isactive("\\$0x332")
```

Syntactically an instance specifier can be used instead of a program name in a TRACE32 **symbol path** (i.e. the path that specifies a variable within the hierarchy given by programs, modules and functions).

The following examples display the **component global variable** `titi`. In the first case it is taken from the instance `adder_instance_1,` where this instance might very well be an instance of a component called adder. The second case only specifies that the variable belongs to component `adder` but makes no statement about the instance. Thus if there are multiple instances, the default instance will be used according to rules outlined in the next section.

```
var.view \\$adder_instance_1\adder\titi  // specifies concrete instance

var.view \\adder\adder\titi              // specifies program,
                                          // displays default instance
```

Auto-Determination of a Component Instance

When displaying the value of a variable, without exactly specifying an instance (e.g. `var.view toto)` the debugger has to “guess” (or be smart) about which instance the user wants to see. The debugger first searches the debug info to resolve the name and find the related debug info. As expected for a module global variable this debug info contains a reference to the THIS pointer (use `y.info` to display it):

```
\\adder\adder\toto
-------------------------------------------------------------------------
   THIS+0009--000B     global frame-relative THIS alive: P:0x0--0x17
-------------------------------------------------------------------------
(global based unsigned int) (unsigned 24 bits)
```

The second step is to choose the appropriate value for the THIS pointer. The default value is that currently stored at `X:0x1` i.e. its “real value”.

However, when displaying variables from non-active components, the debugger needs to deduce the correct THIS pointer by other means. First it tries to resolve the name by searching all debug info (i.e. all the symbols from all loaded component) to find a match. If the matching component (i.e. that component whose debug info was found) was instantiated only once, the debugger will use its THIS pointer and can display the variables value.

If there are multiple instances of the same component, the debugger uses the THIS pointer of the “default” instance. The default instance is marked in the `y.autolist.list` window with a check mark. Note that the concept of default instances is used to arbitrate only between multiple instances of the same component. Therefore there may be many default instances at the same time, if they belong to different components.

The default instance changes when

- the core enters the program code of an instance
- a new instance is created. The latest instance that is found in the load map by the debugger becomes default. Thus the first instance of a component automatically begins its life being default component.
- the user selects a different instance (by “select component” in the context menu in **sYmbol.AutoLOAD.List**)

When an instance is chosen, the debugger uses its THIS pointer to calculate the memory location of the variable and displays its value.

Component Specific and Deferred Breakpoints

As there can be multiple instances of a module, TRACE32 allows to set instance specific break points. Program code is shared between instances, but each instance has a separate data section. The data-section is identified by the so-called THIS-pointer (a system-global variable that points to the data memories used by the component). Therefore, in essence, an instance specific break is implemented as a conditional break (**Break.Set ... /CONDition**) that checks the current vale of the global THIS variable. If the value matches that of the specified instance, the debugger stops the core. If it does not match, it restarts the core again. In any case this constitutes a real time violation.

A breakpoint that is set onto functions that reside in yet unloaded modules is called a deferred breakpoint. Because the debugger cannot write the breakpoint to memory (because the module is not yet loaded) it needs to keep track of loaded and unloaded components and set the breakpoints, when the respective component becomes available in memory.

To realize when a module is loaded or removed, the debugger sets breakpoints on module constructor and destructor code. When these breakpoints are hit, they invoke a script that in fact checks if a module became available. If so, all relevant breakpoints are set by the script. If a module is removed from the system, the debugger deletes the corresponding breakpoints from memory (this is handled internally by the debugger).

The debugger needs to be configured for NMF support. The first step is to enable NMF support via

```
SYStem.Option.NMF ON
```

This is fundamental because it is required so that the debugger reads the current THIS pointer from the target when the core stops.

The next step is to configure the autoload mechanism i.e. tell the autoloader which command or script to execute for each component found in the NMF load map. Typically this is done with a command like

```
sYmbol.AutoLOAD.CHECKNMF "do &nmf_scripts_dir/autoload.cmm NULL"
```

The script will be passed a number of parameters pertaining to the component that is to be loaded and has the objective to load its debug symbols into memory and relocate them as appropriate (e.g. via **Data.LOAD.Elf ... /NoCODE**). By convention the name `autoload.cmm` is used for the script.

Finally there is a command to read the loadmap from target memory, parse it, and call `autoload.cmm` for each newly loaded component:

```
sYmbol.AutoLOAD.CHECK
```

Note that this works only, if the command is invoked **without** parameters. When used with parameters `ON` or `OFF`, it enables `automatic` checking of loadmap changes each time the core stops after a program run. This usage is discouraged for NMF.

```
Data.LOAD.Elf ...  [ /DNMF|/NMF ]  [ /RELOC <sect1> AFTER <sect2>]
```

The command **Data.LOAD.Elf** was extended with the options `/DNMF` (dynamic NMF) and `/NMF` (static NMF) for controlling how variables that are located the relative to the `THIS` pointer are resolved:

- `/DNMF` : the debugger resolves the references to the `THIS` pointer while the variables are displayed so that a change in the `THIS` pointer will immediately show the new memory contents. Also, some duplicated symbols are suppressed to simplify debugging. This option is usually used always when loading an ELF file in the context of NMF.
- `/NMF` : references to the `THIS` pointer are resolved at load time. Therefore, later changes to the `THIS` pointer are not recognized. The parameter normally is not used, because the behavior of the debugger is too "static".

When not using either of the options (default), the behavior is similar to `/DNMF` with the difference, that no duplicate symbols are removed.

With `Data.LOAD.Elf` on MMDSP the option `/RELOC` supports the keywords **`after`** and **`like`**. The expression

```
/RELOC mem1.1 at X:0x316 /RELOC mem1.2 like mem1.1
```

relocates `mem1.1` so that it starts at the address `0x316`, through moving it by a certain distance (which is calculated by the debugger). Using the keyword **`like`** relocates a section through moving it by the same distance that was used in the relocation of the section used as reference, so in the sample section `mem1.2` is moved in the same way as section `mem1.1`. This relocation preserves gaps between sections as they occur in the unrelocated case.

In contrast to `like` the keyword `after` relocates `sectB` so that it will start directly after the end of `sectA`:

```
/RELOC sectB after sectA
```

`After` removes potential gaps (which may be as small as a single word) between sections and thus potentially changes the memory layout. Therefore usually you will want to use the **`like`** keyword.

Autoload Script

This section outlines the usage of the script `autoload.cmm` . The script is called by the debugger after the **sYmbol.AutoLOAD.CHECK** command for each newly loaded component in order to load its symbols into memory.

The script is passed a number of parameters corresponding to the elements of the loadmap. In the case of Loadmap v1.1 these are:

```
&elf_dir &pSolibFilename &pAddrProg &pAddrEmbProg &pThis &pComponentName
```

For Loadmap v1.2 two additional parameters are passed:

```
&elf_dir &pSolibFilename &pAddrProg &pAddrEmbProg\
                                    &pThis &pComponentName &pXROM &pYROM
```

The central part of the `autoload.cmm` script is the call of the `data.load.elf` command to load the new symbols:

```
Data.LOAD.Elf
  &elf_dir/&pSolibFilename         // location of the .elf file
  /NoCODE                          // only load symbols
  /NoClear                         // keep already loaded symbols
  /RELOC mem0.1 at P:&pAddrProg    // relocate program section
  /RELOC mem1.1 at X:&pXROM        // relocate XRom data sec. (ONLY v1.2)
  /RELOC mem2.1 at X:&pYROM        // relocate YRom data sec. (ONLY v1.2)
  /NAME &pComponentName            // name the program
  /StripPART 7                     // remove part of the symbol path
  /dnmf                            // dynamic resolution of variables
```

Helper Functions

There are some helper functions for automating work with component in PRACTICE scripts:

- `nmf.this(component specifier)`: returns the THIS pointer of the given component
- `nmf.isactive(component specifier)`: checks if the given component is active i.e. its THIS pointer matches the current value of the system-global THIS variable (at X:0x1). Returns TRUE or FALSE.

Examples using different formats for specifying a component:

```
print nmf.this("\\$adder_instance_1")
print nmf.isactive("\\$0x3FB")
```

As these functions expect string arguments, the double quotes ""are mandatory and also the prefix "\\$" is required. Without these, the names are not recognized as instance specifiers (specifier in the sense that they can either be a this pointer or an instance name).

Relocation of Sections

Because components can be loaded at arbitrary addresses under NMF, TRACE32 has to relocate the debug symbols and adapt them so that they point to the actual addresses (normally components are compiled as if they were placed at `0x0`). In TRACE32 relocation can be done by relocating individual symbols or groups of symbols with the command **sYmbol.RELOCat**e. The following command will move all symbols in the module `somemodule` (of the program `uniop`) to the program address `P:0x1000`

```
y.reloc p:0x1000 \\uniop\somemodule
```

However, in the context of autoloading it is faster and easier to relocate complete sections, e.g. relocating all program symbols at once instead of each module separately. Section relocation is done by passing one or multiple `/RELOC <section_name>` parameters to the **Data.LOAD.Elf** command:

```
data.load.elf build/_st/api/uniop.elf
    /RELOC mem0.1 AT P:0x1000
    /RELOC mem1.1 at X:500
```

After loading an elf file the names of the sections can be listed with **sYmbol.List.SECtion**

The demo scripts assume the following fixed names for all .elf files.

```
program section     mem0.1
data section        mem1.1
data section        mem1.2   (optional)
```

Usually relocation for NMF is done such that “primary” sections like `mem*.1` are relocated with the `AT` keyword (to an address that is obtained from the loadmap) and “dependent” sections like `mem*.2 mem*.3, mem*.4` are relocated with the `LIKE` keyword as shown in the following example:

```
Data.LOAD.Elf c:/etc/mmdsp/cr/mcr333/ispctl.elf /NAME ispctl
  /NoCODE /NoClear /DNMF
  /RELOC mem1.1 AT X:0x316
  /RELOC mem1.2 LIKE mem1.1 /RELOC mem1.4 LIKE mem1.1
```

Please note:

- When using **Data.LOAD.Elf** with the `/RELOC` parameter (in order to relocate a complete section) one must use only the section name, but never include the program name (shown as "path" in the **sYmbol.List.SECtion** window). When adding the path name, the debugger will not recognize the section and not relocate anything (without warning).
- When *relocating a data section* (in X: or Y: memory) one must add "`X:`" to the **RELOC** parameter e.g. `/RELOC mem1.1 at X:0x1000`". If `X:` is omitted, address calculation fails due to different width of P-mem and X-mem.
- Besides the **command** **sYmbol.List.SECtion** there are some utility **functions** (that return a value) for working with sections: **sYmbol.SECRANGE()**, **sYmbol.SECADDRESS()**, **sYmbol.SECEND()**. Please see the TRACE32 documentation for more information.

Rarely Used Commands

There are some additional commands related to NMF that are hardly used because they were superseded by other mechanism during development of NMF support. These commands are described briefly in this section. For further information please contact Lauterbach support.

- `y.autoload.check ON | OFF`

  Configures the debugger so that it will look for changes in the loadmap always when the core stops after a program run, because the loadmap might have changed. The disadvantage of this approach is that the loadmap would be read frequently and in particular also after each single step.

  For tracking loading and unloading of modules it is instead recommended to set conditional breakpoints onto the `Constructer_Run` and `Destroyer_Run` symbols and have them execute scripts that update the loadmap.

- `data.load.elf .... /NMF <thisptr>`

  When using data.load.elf with the /NMF parameter, the debugger will resolve all references to the THIS pointer when loading the debug info by assuming the argument as value for the THIS pointer. Thus the debug info does not contain the reference to THIS anymore (see below).

  The problem with this approach is that it cannot handle well multiple instances of a component.

```
y.info \\adder_instance_2\adder\m
-------------------------------------------------------------------------
X:00033B--00033B     global static
-------------------------------------------------------------------------
(unsigned int) (unsigned 24 bits)
```

Various notes

This section contains various comments related to the debugging of NMF.

- For complete configuration examples see demo files supplied by STM.
- The "program name" or component name must not be identical between different files. In the context of NMF debugging this is assured by the convention of creating a program name from the file paths and the .elf file name (e.g. `dec_one$adder`). If there are identical program names, it is undefined which program is used.
- `Warning: stackframe overlap error (0x18) in file add`
  This warning means that the debugger encountered conflicting stack frame information. Usually this happens when loading multiple .elf files and not relocating them properly so for the same P-mem range multiple frame records are found.
- You can get a list of the frames by **sYmbol.List.FRAME**. Each entry tells the debugger for a certain program range where registers are saved (e.g. relative to the current sp0).
- It is not possible to set a deferred breakpoint onto the very first NMF component to be loaded. The reason is that the NMF-framework only sets up the execution engine and loadmaps at the time when loading the first component. Therefore the MMDSP-debugger cannot set the pending breakpoint before this.
  The obvious fix is that NMF-framework instantiates the EE and the loadmap already when starting up. However, according to STM, the fix will not be made soon.