Software Analysis in CodeWarrior for MCU
Basic Concepts

by: Razvan Ionescu
DevTech. Software Engineer
Bucharest
Romania

1 Introduction
Embedded engineers use features like debugger and compiler, and rarely use Software Analysis features like Trace, Profiler, or Code Coverage. However, those tools are less known but are as important and help a lot in a project. CodeWarrior for MCU provides an extensive set of Software Analysis (SA) tools, this application note explains and popularizes SA basic concepts. It will demonstrate how to enable Trace and Profile, how to collect trace, and how to read the results. ARM based processor from Freescale, Kinetis K70, and CodeWarrior for MCU v10.2 (b120126) are used to demonstrate SA capabilities.

2 Theory of Operation
There are some basic SA concepts that need to be explained before presenting main SA features in CodeWarrior.

2.1 Internal vs. external buffers
The buffer is the location where an executed trace is written by the hardware during the application. Most of the time, buffers are implemented using RAM memory. Buffers can be located on the analyzed device (internal) or on a trace
collector device (external), which is usually used to perform the run-control function. CodeWarrior supports for Kinetis one internal buffer and two devices with external buffers (P&E TraceLink and Segger J-Trace).

![Diagram of Kinetis device with internal and external buffers]

**Figure 1. Internal and external buffers**

For the Kinetis family, platform 2 and platform 3 type devices support internal buffers and complete external buffer implementation. The size of internal buffer is 2 kB. The internal buffer is called Embedded Trace Buffer (ETB). The size for external buffer is 128 MB (P&E TraceLink) or 8 MB (Segger J-Trace).

### 2.2 Hardware trace capabilities for Kinetis

The Kinetis family has support for program trace (Embedded Trace Macrocell (ETM) and instrumentation trace (Instrument Trace Macrocell (ITM)) for platform 2 and 3 devices. Platform 0 and 1 devices has support only for ITM trace.

The program trace (ETM) provides information about executed code in a compressed format that allows, after an offline post-processing, to fully reconstruct the application execution. It is possible to use methods to limit the amount of program trace generated by the core.

![Diagram of program trace (ETM)]

**Figure 2. Program trace (ETM)**
Instrumentation trace (ITM) streams information from two sources: application code and Data Watchpoint and Trace (DWT) unit. A developer can insert into application code (to instrument) that writes to certain registers with custom data. That data is inserted into the ITM trace in raw format. From the DWT, some kind of data trace is generated. Performance counters values are also sampled into the ITM trace.

Figure 3. Instrumentation trace (ITM)

The Trace Port Interface Unit (TPIU) and Embedded Trace Buffer (ETB) are formatters, and package the trace. Trace funnel has the role of mixing the trace streams from the ETM and ITM in one stream.

Figure 4. Trace formatters—ETB and TPIU

NOTE

Kinetis does not allow to send trace simultaneously to the ETB and TPIU. Only one path can be used at once.
2.3 Timestamps

There are 2 timestamp counters for Kinetis: a global 48-bits counter and a local 21-bits counter. The global counter can be used to put timetamps on events in ETM and ITM trace streams. By doing this, you have a synchronization of events from the 2 streams, that is because the clock is the same. The local timestamp counter can be used only by ITM events.

Also note that the global timestamp counter does not stop while you are in debug mode, so the time spent in debug mode is also recorded (for example, breakpoints, and prints). The local timestamp counter is frozen during debug mode, therefore it is counting only the time spent in execution mode.

2.4 Trace collection modes

There are three types of collecting trace modes:

- **Overwrite**

  In this mode, when trace events fill the trace buffer, the first events (oldest) are overwritten by the last events (more recent). At the end of the collection, the trace buffer will have the last n events, where n is trace buffer capacity (calculated in events). Think of this as a circular buffer.

  This mode is useful for when the user investigates an extreme condition (for example, a crash or an exception), because in trace the last event is found that generated that condition.

  A special attribute of this mode is that it is the least intrusive—basically trace IP monitors core/platform execution, and record events, but do not interfere with execution.

  This mode can be used with ETB (internal trace buffer) or with an external buffer (P&E TraceLink or Segger J-Trace).

- **Continuous**

  In this mode when the trace buffer is full the core stops execution and enters in debug mode. CodeWarrior detects debug status of the target and collect trace and then resumes execution. This mode assures that all execution is traced and kept and nothing is lost. However, it is intrusive and may break the real-time nature of an application.

  This mode is useful when a Profiler is needed (to gather enough data to make the Profiler useful). Also it is useful in case of Code Coverage analysis, where real-time is not a strong requirement.

  This mode can be used only with the internal trace buffer (ETB).

---

**Figure 5. Overwrite trace**

- Trace is not full
- Trace is full
- Trace overwrites oldest records
Figure 6. Continuous trace

- **Pseudo-continuous**
This mode is a combination of the first two. It consists of collecting trace with an external device (P&E TraceLink or Segger J-Trace) and keeps all buffers. However, if an external buffer (which runs by design in overwrite mode) overflows, then a gap appears in trace.

Figure 7. Pseudo-continuous trace
2.5 Profiling counters

Kinetis supports five 8-bits profiling counters and one 32-bits cycles counter:

- CPI counter (8-bit):
  
  This is the general counter for instruction cycle count estimation. It increments on any additional cycles (the first cycle is not counted) required to execute multi-cycle instructions except those recorded by LSU counter. It also increments on any instruction fetch stalls.

- LSU counter (8-bit):
  
  It is the load-store counter. The counter increments on any additional cycles (the first cycle is not counted) required to execute multi-cycle load-store instructions.

- FOLD counter (8-bit):
  
  It is the folded instruction counter. The counter increments on any instruction that executes in zero cycles.

- Exception counter (8-bit):
  
  It is the exception overhead counter. The counter increments on any cycles associated with exception entry and return.

- SLEEP counter (8-bit):
  
  It is the sleep overhead counter. The counter increments on any cycles associated with power saving, initiated by the WFI or WFE instructions.

- CYCLES counter (32-bits):
  
  It is the cycles counter.

2.6 Single Wire Output (SWO) support

Single Wire Output (SWO) is a protocol that outputs the ITM trace through a separate path. This is basically a low-cost debug and trace method. The bandwidth is limited, but enough information can be sent to create a big picture of application execution. This type of trace collection does not need of a trace buffer (internal or external) and uses a streaming approach. All data is sent directly to the host machine.

This method of tracing is not intrusive.

**NOTE**

SWO can be used only over SWD protocol (not over JTAG), therefore make sure debugger protocol is properly set.

2.7 How to instrument an application

Kinetis provides 32 stimulus registers. When data is written to one of these registers, it is automatically inserted in trace. A common code that maps a register and writes data is:

```c
volatile unsigned long *itm_stim_0 = (unsigned long *) 0xE0000000;
...
*itm_stim_0 = 0x11223344;
...
*itm_stim_0 = 0x55667788;
```
2.8 Trace protocol synchronization packets

Both ETM and ITM trace streams implement a set of synchronization packets that allow to recover after a special condition (for example, trace lost, and trace buffer overflows).

Synchronization packets can be:

- **Periodic (ETM):**
  
  After each 1024 bytes entered in ETM trace, a sync packet is sent. This is useful, for example, when you have trace collected using a circular buffer, ETB or external device, in overwrite mode. After the buffer is full, the oldest records are overwritten with the most recent ones. In this case, it will overwrite sync packets (sent at the start of trace) and the stream will be un-decodable. After 1024 bytes, a periodic ETM sync packet is sent, the stream will then be correctly decoded from that point.

- **Periodic (ITM):**
  
  The default value for the ITM sync packet is 1024 bytes. This can be adjusted. Please see the above description to see how the sync packet helps recreate the execution flow.

- **Address sync (ETM):**
  
  Even if a periodic sync is emitted by the ETM from 1024 to 1024 bytes, this is not enough to recover from a trace lost or from a trace buffer overwrite. The stream is decoded, but ETM compression mechanism for code addresses is made with relative offset. In other words, after the ETM places a packet in address 0x100, if the next executed address is 0x104, the ETM codes the address as 0x4. This allows to save bandwidth, but generates some wrong addresses to be decoded after a special condition event (for example trace lost, and trace buffer overwrite). But an Address-sync packet will sample a full address value into the trace. From that point, both decoding and trace recreation is correctly executed.

- **Timestamp sync (ETM):**
  
  As for instruction addresses (see above), the timestamp value is also coded using a relative offset. So just after a timestamp sync, the time reported in trace is accurate after a special condition event (for example, trace lost, and trace buffer overwrite).

- **General trace sync (ETM, ITM):**
  
  There is a special packet that differentiates the start of the trace from noise. This is put in trace on events like: start trace, exit from debug, or at other events that may trigger a trace start.

- **Trigger packets (ETM, ETB):**
  
  Are not sync packets from a protocol perspective, but can be viewed as logical sync points. It marks the tracing triggering point.

2.9 Extra trace features in Kinetis

There are some other trace, or trace related capabilities in Kinetis, but they are not the object of this application note.
### Practical Content

This section indicates how to import a project into CodeWarrior, enable Trace & Profile, instrument the code, run application on K70 target, collect trace and see the results. All trace collection modes are shown (overwrite, continuous, and pseudo-continuous). The SWO collection is also shown.

1. Choose trace collection mode: overwrite, continuous, pseudo-continuous, and SWO
2. Run application (collect trace progress monitor)
3. SA view
4. Trace
5. Critical code and coverage
6. Performance and call tree
7. Log
8. Repeat c-i for all trace collection modes

### 3.1 Import the demo project

1. Open CodeWarrior and right-click in the projects area.
2. Click on Import, then choose to open an existing CodeWarrior project:

![Import menu](image)

**Figure 8. Import menu**

2. Click on Import, then choose to open an existing CodeWarrior project:
3. Browse to the project location:
Figure 10. Browse to project location

4. Select project’s directory:
5. Notice the project is recognized by CodeWarrior; click Finish:
6. Build the project.

3.2 Instrument the code

Open main.c file and edit as below:

```c
void Performance1(void);
int Recursive(int);

volatile char iteration = 0;
volatile unsigned long *itm_stim_0 = (unsigned long *) 0xE0000000;

void Launch(FUNC_TYPE f, int arg)
{
    #pragma no_inline
    f(arg);
}

void InterruptTest();
```

Figure 13. Edit main.c file – define stimulus port
Define the stimulus port to write the custom data. Then write some custom data from code:

```c
void main(void) {
    *itm_stim_0 = 0x11223344;
    Performance1();
    Recursive(3);
    Performance1();
    *itm_stim_0 = 0x55667788;

    for (;;) {
        entry();
        *itm_stim_0 = 0x87654321;
    } /* loop forever */
    /* please make sure that you never leave main */
}
```

Figure 14. Edit main.c file – write to stimulus port

### 3.3 Enable Trace and Profile

Open Debug configurations for imported project and select tab *Trace and Profile* for k70_appnote_MK70FN1M0_INTERNAL_RAM_PnE U-MultiLink configuration.

![Select Trace and Profile tab](image)

Figure 15. Select Trace and Profile tab

Check Enable Trace and Profile and make sure ETB, Continuous Trace Collection, Collect Program Trace, Collect instrumentation trace, and Collect Profiling Counters option is checked.
Click Apply to confirm settings.

Click on Advanced Settings to see some more options:
Figure 17. Enable and configure ITM tracing

Notice that ITM tracing is enabled. Synchronized packages are default on, because the bandwidth limitation is negligible to adding them and the benefit is huge. This helps synchronize the trace in case of a special condition event (for example, trace lost, and trace buffer overwrite). Notice that Local timestamp is used for this example.

Here only 1 stimulus register (from the application, write only at this register) is enabled. Also note the Performance counters enabled (section Event Generation). This triggers counter values to be sampled in trace.
For ETM advanced settings, notice the Stall processor option. This freezes the core activity when the internal ETM buffer is full to not lose trace. However, this function is limited because of many reasons, however it is better to have this option on by default. The trace all branches option configures the ETM trace to record all branches or just indirect branches. Theoretically, it is enough to sample only indirect branches, because other branches can be determined offline, just disassembling the binary executable. It is a method to save bandwidth. However, if you don’t have overflows in trace due to bandwidth capacity, it is advisable to check this option for better decoding results.

Enable timestamps option configures trace to sample or not the timestamp. It is another method for saving bandwidth, however it means that trace will not have timestamps, so some analysis, like performance or critical code cannot be performed. If you want to put more code execution information in trace and you are not interesting in the timestamp, this option saves bandwidth.
3.4 Collect trace

Enter debug mode and resume application. Notice that the current settings (continuous mode), the trace module stops the target execution when the internal buffer hits threshold, collect trace, and then resumes execution. In the bottom-right corner of CodeWarrior, look at the progress monitor for collecting trace operation.

![Collecting trace progress monitor](image)

**Figure 19. Observe collection of trace progress monitor**

The debugger has two buttons that control execution stop: Suspend (circled in picture below) and Terminate (indicated by an arrow in picture below).

![Debugger controls](image)

**Figure 20. Debugger controls**

When Terminating an application, means that you want to exit immediately, and trace will not be collected then. However, for continuous mode, you will have trace collected up to the last collection moment. For overwrite mode (where trace is collected only when the application is suspended) you will have no trace.

In this case, Suspend and then Terminate application.

The Software Analysis view for results opens automatically.

3.5 Software analysis view

Software Analysis view opens automatically after trace collection. However, if you want to open this view on demand, go to the CodeWarrior menu: Window->Show view-> Software Analysis.

![Software Analysis view](image)

**Figure 21. Software Analysis view**
Notice buttons on the upper-right side of the view. You can refresh the view, expand or collapse results.

Right-clicking on a result set, you can delete or save it.

![Save/Delete results](image1)

**Figure 22. Save/Delete results**

3.6 Trace view

Trace view shows the collected trace.

![Trace viewer controls](image2)

**Figure 23. Trace viewer controls**

Notice trigger packets (ETB and ETM), synchronization packets, drop packets (due to missed synchronization for ITM stream), function symbol name (for example, main), and timestamps (they are different for ITM and ETM events, because they use two different clock bases).

See the buttons on the upper-right side of the view, they allow to easily navigate in trace, expand trace events to see assembly and source code mixed in the same view, find a record, export to .csv format, and configure timestamp resolution.
Figure 24. Events in trace

Notice the event for a performance counter (Cycles counter overflowed) and also events for a branch, where both source and destination of the branch are displayed.

Figure 25. Mixed trace (source and assembly code)

By pressing on the Expand all button, you can see assembly and source code mixed in trace.
Figure 26. Find in trace

You can search for an event in trace. In picture above, see the special event Trace buffer read finished. This appears only for continuous mode collected trace and signal when a trace is full and is uploaded on the host. Notice that after this event (which means entering in debug mode for the target), trace packets for resynchronization are sent by the ETM and ITM modules.

3.7 Timeline view

The timeline view shows a logic analyzer like view of an application execution.
Figure 27. Timeline view

Notice that you have two cursors and you can make measurements on this view. On the left side, the executed function is shown together with the percentage from the execution time.

Figure 28. Timeline cursors
3.8 Critical code view

The critical code shows a flat profiler (a profiler that counts each function execution by summing the individual instruction time; this is different from an hierarchical profiler presented in next section) and code coverage.

You can configure what to see in the upper or bottom panels:

Notice the mixed view (assembly and source code) and statistics for every executed line. You can choose to express the values using colored bars to easily spot biggest time.

**NOTE**

Bar length is not scalable at the percentage of each execution. Instead, the biggest time has a bar showing 100% and the rest are relative. This is done to avoid little values to not be visible.
Figure 31. Critical code – mixed source code view

You have the option to show code only in assembly:

Figure 32. Critical code – assembly code view

Or only as C source code:
3.9 Performance view

The performance view is a hierarchical profiler (it shows functions statistics calculated as parent-child or caller-callee pairs). It presents self-time (exclusive time) for a function (time spent only in that function) and hierarchical time (inclusive time) for a function (time spent in that function and in all its children functions).

3.10 Call Tree view

The Call Tree view shows functions calling the tree together with some statistics.
Notice the dark grey nodes; they indicate the critical call chain, this means the chain of functions that consume most of the time.

3.11 Log view

If it is enabled, the log will display the interaction of CodeWarrior with the target. You can see how intrusive was the collection of trace or you can analyze your activity (debugging) for an application execution.

![Log view](image)

Process stopped at 19:10:22 2/2/2012
Process resumed at 19:11:14 2/2/2012
Process resumed at 19:11:19 2/2/2012
Process stopped at 19:11:24 2/2/2012
Process stopped at 19:11:24 2/2/2012
Process resumed at 19:11:28 2/2/2012
Process stopped at 19:11:33 2/2/2012
Process resumed at 19:11:37 2/2/2012
Process stopped at 19:11:42 2/2/2012
Process stopped at 19:11:43 2/2/2012
Process terminated at 19:11:47 2/2/2012

![Log view](image)
3.12 Trace in overwrite mode

The figure below shows how to configure trace collected in overwrite mode.

You need to make sure that Continuous Trace collection option is not checked. Overwrite collection mode is the default mode.

Please download the application and run it. Suspend it at some point, then Terminate application. Open Trace view.
Figure 38. Dropped packets in trace

Notice that start of trace has a lot of dropped packets. This is because in overwrite mode, trace has big chances to overwrite the synchronization packets.

Figure 39. Partial trace synchronization in overwrite mode

See above how an ETM synchronization packet obligates the decoder to correctly recognize the ETM packets. However, notice that logic function is not yet restored, since at this point addresses and time is relative to 0. An address and a timestamp synchronization packet is needed to correctly decode the stream.
Figure 40. Complete trace synchronization in overwrite mode

See above how a synchronization packet for address makes trace to correctly decode the functions and later, timestamp sync packets (not specifically shown in trace) display the correct timestamp (see the jump from 6,008 to 5,560,176,891,786 cycles). From this point, the trace is perfectly synchronized, both for address and timestamp.

3.13 Trace collected with an external device

An external device (Segger J-Trace or P&E TraceLink) can be used to collect trace in overwrite mode, but with a bigger buffer (internal buffer ETB has 2 KB, external device J-Trace has 8 MB, external device TraceLink has 128 MB).

NOTE

The internal bus from ETM and ITM to Trace funnel (common sink, that mixes the two trace streams in one stream) is 8-bits wide. The path from trace funnel to ETB is on 32 bits, so it allows to buffer packets that come at great speed. That’s why, ETB has the lowest rate of lost trace.

From Trace funnel to TPIU, the bus is on 8 bits. From TPIU to TPA (Trace Port Adapter) where external devices are plugged, the bus can be set to 1-bit, 2-bits or 4-bits. Notice that this is a bottle-neck and causes losing trace when it comes with a high frequency rate.

This is how to set trace collection to use the Segger J-Trace external device:
With J-Trace, Continuous Trace Collection is grayed, because only the overwrite mode can be performed. However, you have Keep all trace buffer option that enables the Pseudo-continuous collection mode. In this mode, whenever suspend application execution trace is collected and kept in CodeWarrior. At next suspend, the new trace is appended to the old one.

Download application on target and set a breakpoint as shown below. We do this to not collect too much trace, because decoding a large trace takes some time. To demonstrate Software Analysis basic concepts it is ok to collect a small amount of trace.

```c
for (;;) {
    entry();
    *itm_stim_0 = 0x87654321;
} /* loop forever */
/* please make sure that you ne
```

Resume application and when it stops in breakpoint, Terminate application. Open Trace view.
Notice that the trace starts from the first record, there are no more triggers in trace. Also remark the synchronization packet that is put by trace after a trace lost (tracing restarted after overflow). This overflow is ETM internal overflow, because the external device cannot read the whole trace at the same rate it is generated.

3.14 SWO trace

The SWO trace is a light trace. Only ITM trace is recorded this way. SWO requires the debug to be executed over SWD interface. In this paper, a Segger J-Trace external device was used to collect SWO trace.

See below how to change the debugger option from using JTAG to use SWD.
Open Trace and Profile settings and configure SWO clock.

![Configure trace collection with SWO](image)

**Figure 45. Configure trace collection with SWO**

Notice that you need to provide a core clock to be used by SWO protocol. This is not the speed of the core, but it is somehow derived from it. For a K70 device (MK70FN1M0) this speed is 10 MHz.

Download the application on target and resume execution. Suspend application and notice that the trace is uploaded and decoded.

![SWO trace](image)

**Figure 46. SWO trace**
Notice that the values write from your code (by instrumentation of the code) are in trace (for example, 0x11223344, 0x87654321).

You can choose to not trace for Performance counters to free some bandwidth.

![Figure 47. Configure collection without Profiling Counters](image)

The trace contains only instrumentation packets.
4 Conclusions

Using the Software Analysis feature from CodeWarrior will help to better understand your application, to identify root causes for various issues (for example, application crash, and exceptions) and to optimize your application. There are plenty of features and ways to investigate your application. Choose your best fitted Software Analysis feature for your project and for your goal.