



Advanced Debugging for CortexTM-M Microcontroller

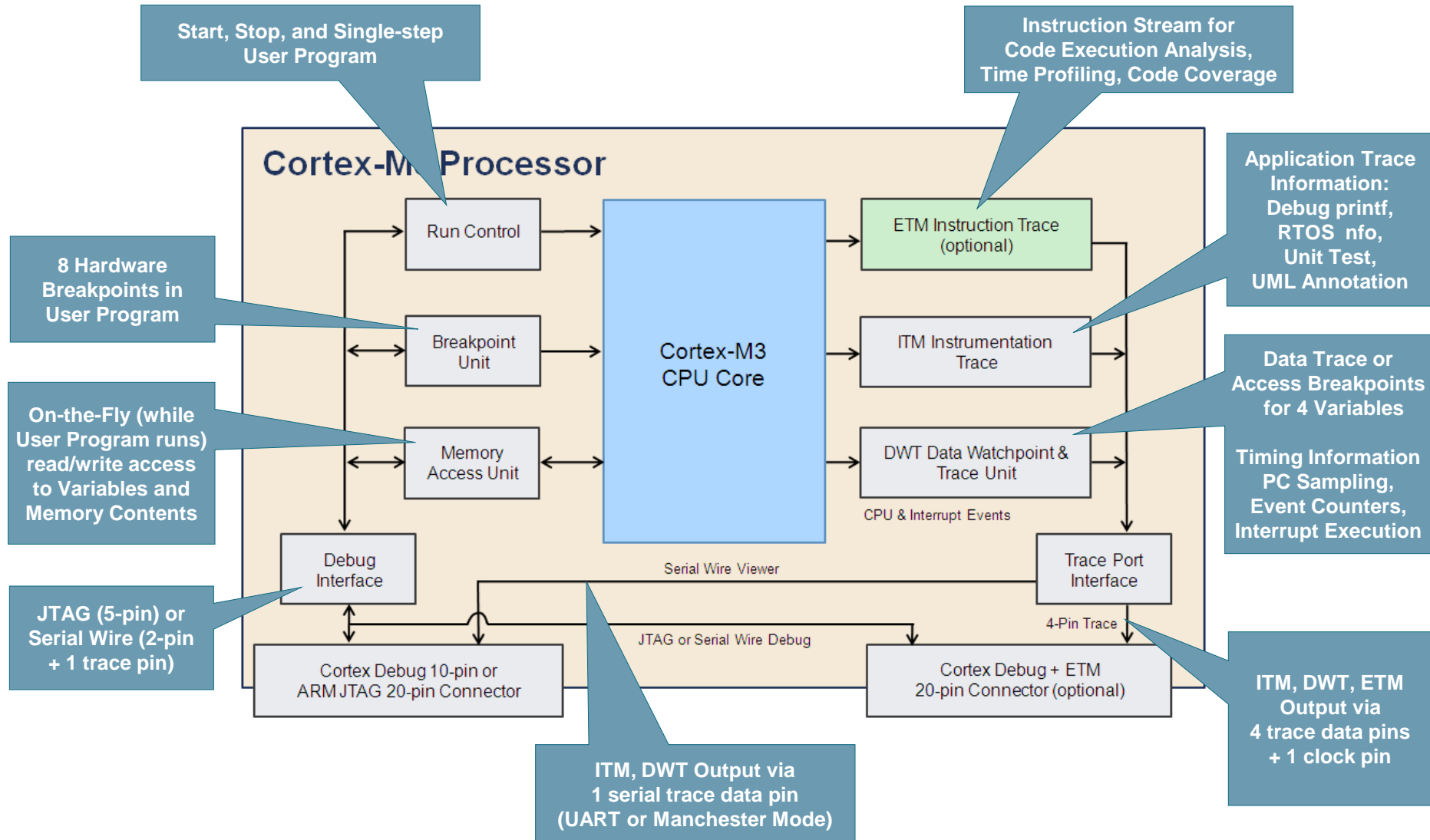
Reinhard Keil
Director MCU Tools



Agenda

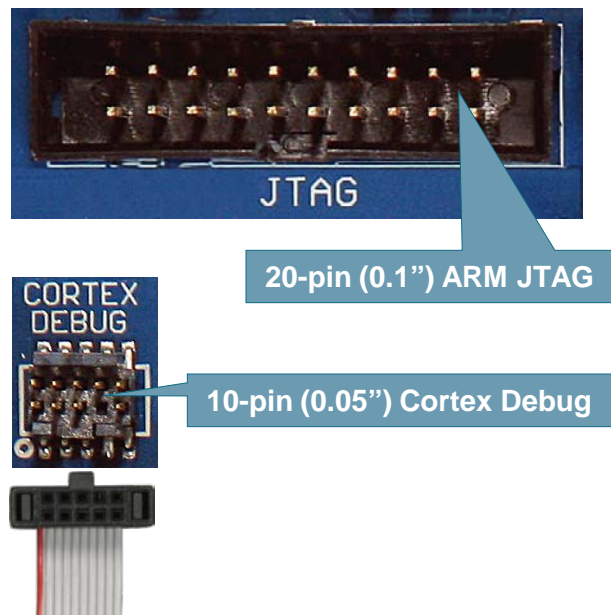
- CoreSight™ Debug Technology for Cortex-M MCUs
 - Debug and Trace Components of a Cortex-M Microcontroller
 - Debug and Trace Connectors
 - Debug and Trace Adapters and Streaming Trace Concept
- Using Debug and Trace
 - Debugging Systems without Stopping Code Execution
 - Trace Record Displays
 - ITM: Instrumented Trace with Code Annotations
 - DWT: Exception Trace, Timing Information, Data Access Trace
 - ETM: Instruction Trace for Code Execution Analysis
- Development Tool Demonstration

Coresight™ Debug Technology (Cortex-M)



Trace (ETM, ITM, DWT) not available on Cortex-M0

Debug and Trace Connectors

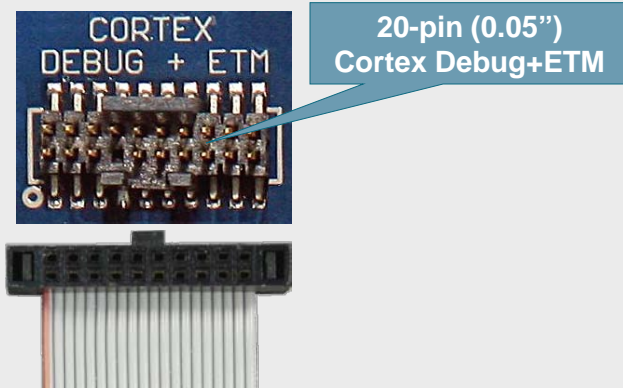


20-pin (0.1") or 10-pin (0.05") Connector

- Identical Debugging capabilities

Support 2 Operating Modes:

- Standard 5-pin JTAG mode (device chaining)
- Serial CoreSight mode
 - 2-pin **Serial Wire Debug** (SWD)
 - 1-pin **Serial Wire Trace Output** (SWO) for Data Trace at minimum system cost

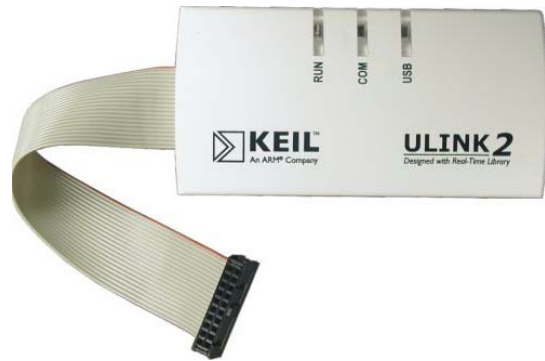


20-pin (0.05") Debug+ETM Connector

- Superset of 10-pin 0.05" Connector
 - Adds 4 (trace data) +1 (trace clock) pins for high-speed Data + Instruction Trace in any operating mode (JTAG or SWD)

More Information: www.keil.com/coresight/connectors.asp

Debug and Trace Adapters



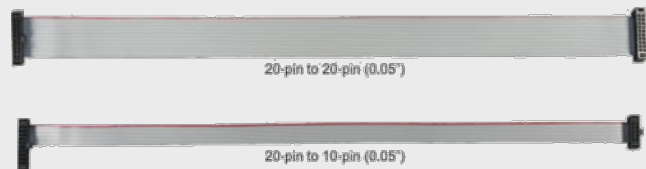
ULINK2: Debug + Serial Wire Trace

- Flash Programming + Run-Control
- Memory + Breakpoint (access while running)
- Serial Wire Trace Capturing up to 1Mbit/sec (UART mode)



ULINK*pro*: adds ETM + Streaming Trace

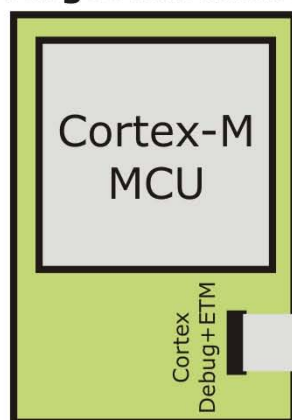
- Cortex-M processors running up to 200MHz
 - 50MHz JTAG clock speed
 - Serial Wire Trace Capturing up to 100Mbit/sec (Manchester Mode)
 - ETM Trace Capturing up to 800Mbit/sec
- Virtually un-limited Trace Buffer
 - Streaming Trace allows complete Code Coverage and Performance Analysis



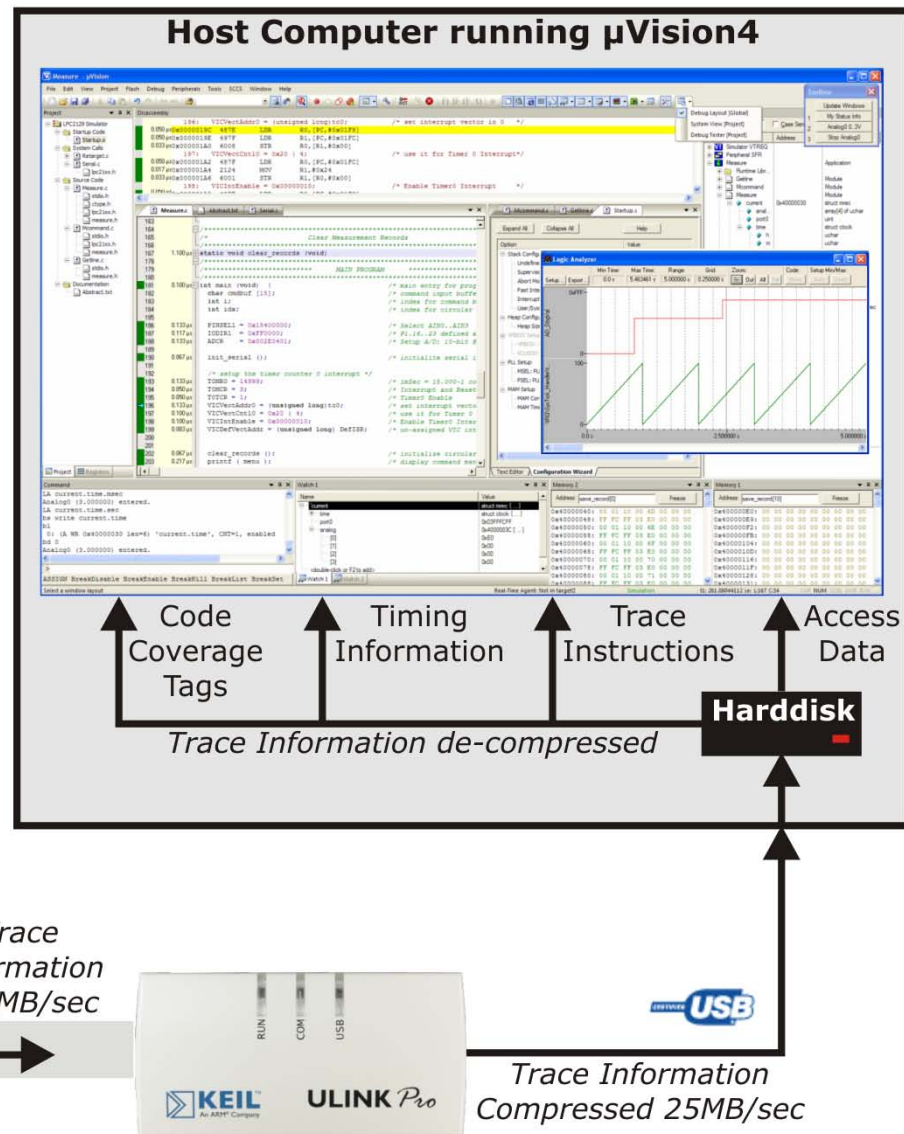
What is Streaming Trace?

- Trace data transferred in real-time to debug host
- Capture size only limited by host resources (harddisk)
- Trace for minutes, hours, or longer
- Required for full code-coverage and timing analysis
- Today's workstations can present trace data instantly

Target Hardware



Trace
Information
100MB/sec



Using Debug and Trace



Run-Stop Debugging has Limitations

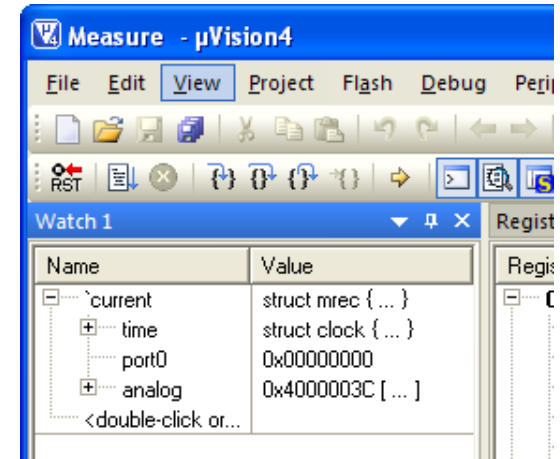
- Stopping code execution changes system behaviour
 - Execution timing cannot be analyzed
- Not servicing interrupts may have catastrophic results
 - Buffers may overrun or connected hardware gets no service
- Many practical problems result from a run-stop debugging
 - Communications systems get into timeout state
 - Motor controllers freeze in high current state



Cortex-M CoreSight Offers Simple Solutions

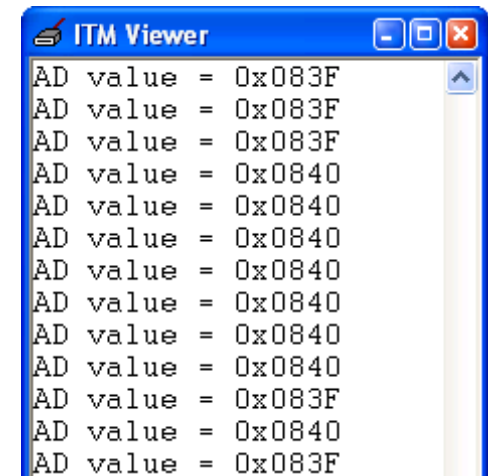
■ #1: Direct memory access to running system

- Native support in the debugger interface
- Values updated dynamically
- In-place editing of live variables
- Breakpoints can be set while system running
- No software overhead, no extra hardware, works with any Cortex-M device!



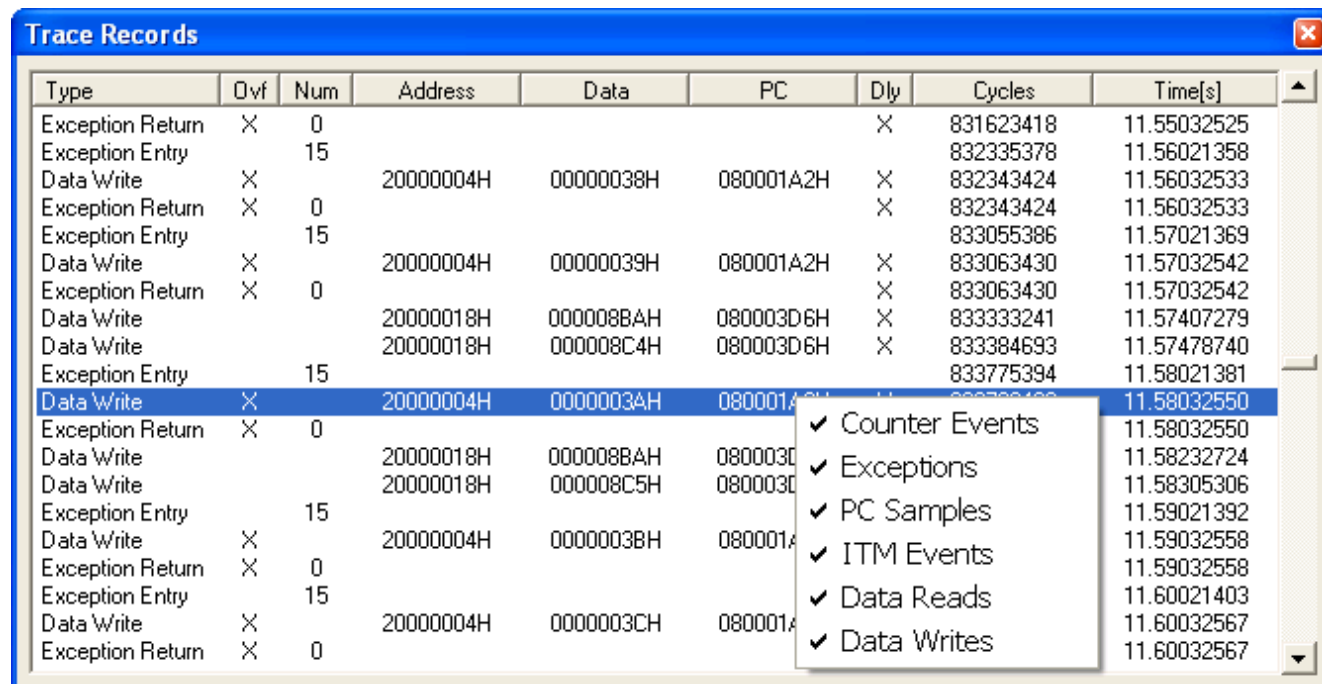
■ #2: printf-style output via an ITM Channel

- Output details to a debug console
- Uses CMSIS standard interface
- Intrusive (the debug printf is a routine that adds to execution time and needs memory), but can be shipped within the end product



Trace Records (DWT + ITM)

- Trace Records display program flow
 - Capture timestamp, PC sample, and Read/Write accesses
 - Time delay and lost cycles are noted
- Raw trace data from all trace sources
 - Filter window to refine the view
 - Updated while target system is running



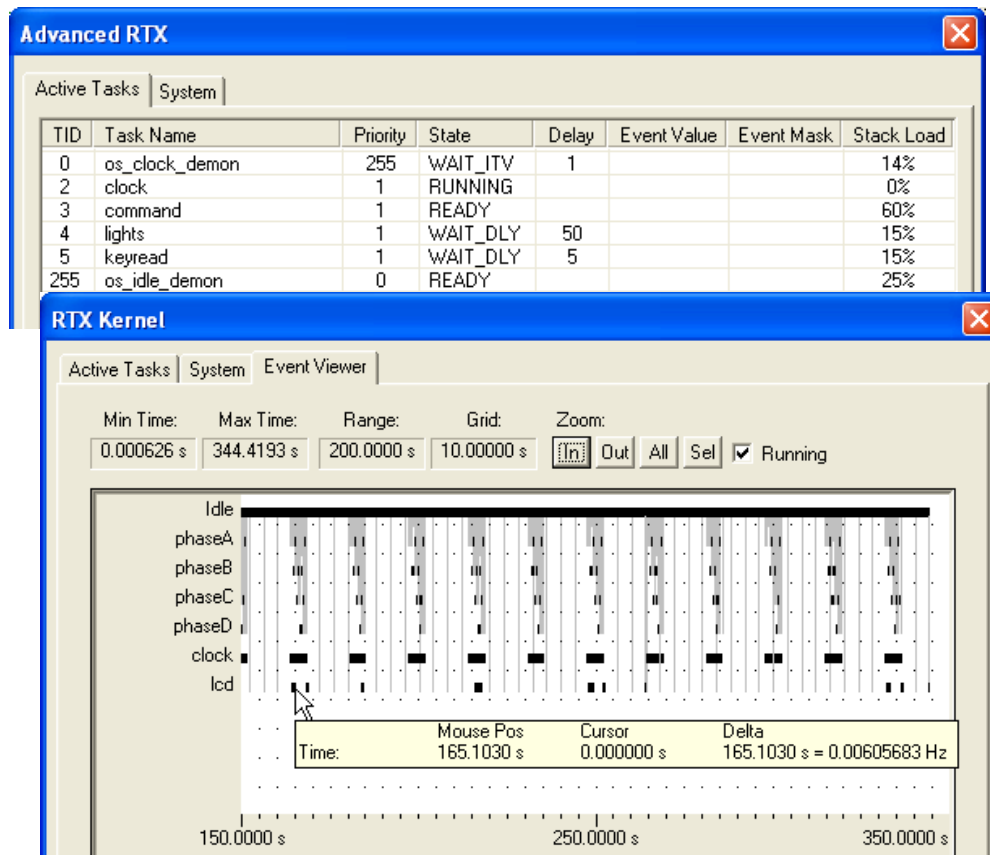
The screenshot shows a window titled "Trace Records" with a table of trace data. The table has columns: Type, Ovf, Num, Address, Data, PC, Dly, Cycles, and Time[s]. The data includes various events like Exception Return, Exception Entry, Data Write, and Data Read. A filter menu is open over the table, showing checkboxes for Counter Events, Exceptions, PC Samples, ITM Events, Data Reads, and Data Writes, all of which are checked.

Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Exception Return	X	0				X	831623418	11.55032525
Exception Entry		15					832335378	11.56021358
Data Write	X		20000004H	00000038H	080001A2H	X	832343424	11.56032533
Exception Return	X	0				X	832343424	11.56032533
Exception Entry		15					833055386	11.57021369
Data Write	X		20000004H	00000039H	080001A2H	X	833063430	11.57032542
Exception Return	X	0				X	833063430	11.57032542
Data Write			20000018H	000008BAH	080003D6H	X	833333241	11.57407279
Data Write			20000018H	000008C4H	080003D6H	X	833384693	11.57478740
Exception Entry		15					833775394	11.58021381
Data Write	X		20000004H	0000003AH	080001A2H		833775394	11.58032550
Exception Return	X	0						11.58032550
Data Write			20000018H	000008BAH	080003D6H			11.58232724
Data Write			20000018H	000008C5H	080003D6H			11.58305306
Exception Entry		15						11.59021392
Data Write	X		20000004H	00000038H	080001A2H			11.59032558
Exception Return	X	0						11.59032558
Exception Entry		15						11.60021403
Data Write	X		20000004H	0000003CH	080001A2H			11.60032567
Exception Return	X	0						11.60032567

- ✓ Counter Events
- ✓ Exceptions
- ✓ PC Samples
- ✓ ITM Events
- ✓ Data Reads
- ✓ Data Writes

Instrumented Trace (ITM)

- 32 ITM channels: write to memory location creates trace data
 - Channel 0: for printf-style debug information
 - Channel 31: for RTX event viewer



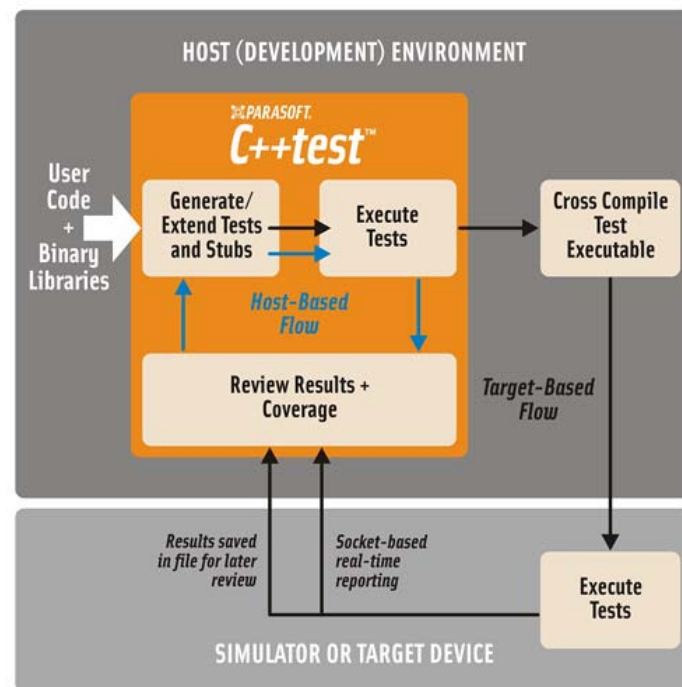
- Remaining ITM channels for user data output

```
// Output 32-bit variable  
// to ITM channel 1  
ITM->PORT[1].u32 = value;
```

A write operation to an ITM channel memory location creates a trace record with output value, time stamp, and program location.

Detailed Code Analysis (ITM)

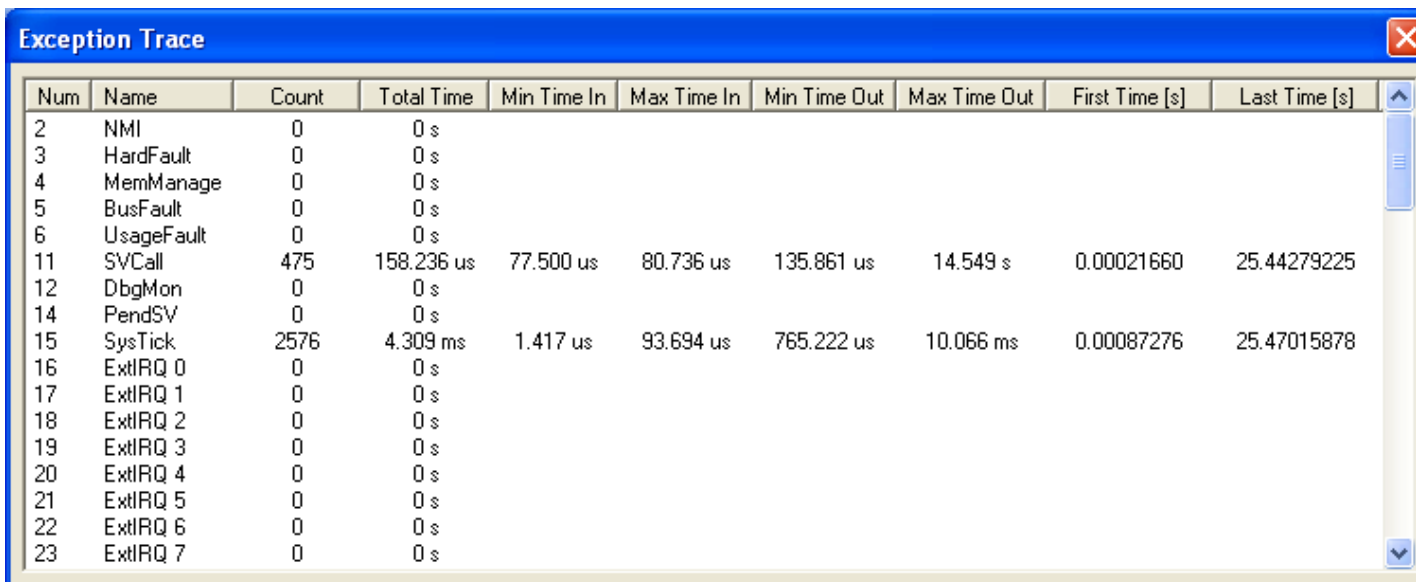
- Parasoft C++ Test™
- Complete C/C++ quality solution for:
 - Static code analysis and coding policy enforcement
 - Automated code review
 - Automated unit and regression testing
 - Host and target test execution
 - Coverage analysis
- Integrated support with MDK-ARM
 - Based on ULINKpro streaming trace
 - Annotated code uses ITM channel for unit test result feedback
- More information:
 - Parasoft Booth: 11-202



C++test's customizable workflow allows users to test code as it's developed, then use the same tests to validate functionality/reliability in target environments

Exception and Interrupt Trace (DWT)

- Statistical information about exceptions and interrupts
- Captures detailed information
 - Name and number of exception; number of times entered
 - Max and Min time spent in and out of exceptions

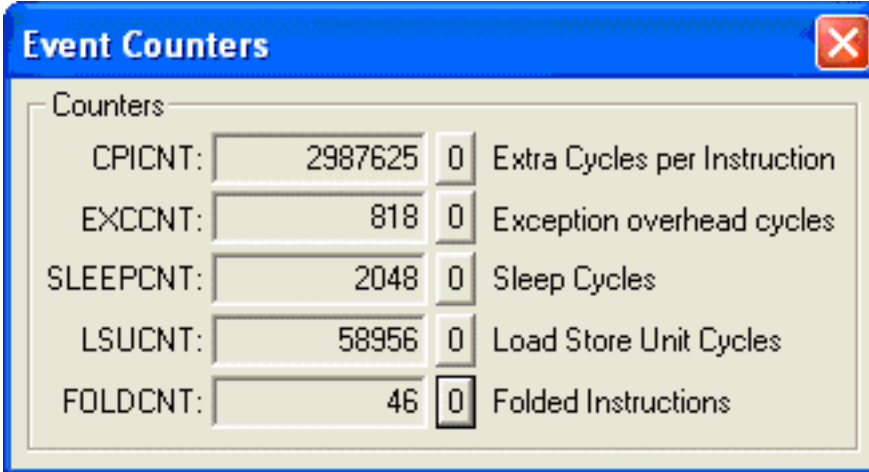


The screenshot shows a window titled "Exception Trace" with a table of exception statistics. The table has columns for Num, Name, Count, Total Time, Min Time In, Max Time In, Min Time Out, Max Time Out, First Time [s], and Last Time [s]. The data is as follows:

Num	Name	Count	Total Time	Min Time In	Max Time In	Min Time Out	Max Time Out	First Time [s]	Last Time [s]
2	NMI	0	0 s						
3	HardFault	0	0 s						
4	MemManage	0	0 s						
5	BusFault	0	0 s						
6	UsageFault	0	0 s						
11	SVCall	475	158.236 us	77.500 us	80.736 us	135.861 us	14.549 s	0.00021660	25.44279225
12	DbgMon	0	0 s						
14	PendSV	0	0 s						
15	SysTick	2576	4.309 ms	1.417 us	93.694 us	765.222 us	10.066 ms	0.00087276	25.47015878
16	ExtIRQ 0	0	0 s						
17	ExtIRQ 1	0	0 s						
18	ExtIRQ 2	0	0 s						
19	ExtIRQ 3	0	0 s						
20	ExtIRQ 4	0	0 s						
21	ExtIRQ 5	0	0 s						
22	ExtIRQ 6	0	0 s						
23	ExtIRQ 7	0	0 s						

Event Counters (DWT)

- Display real-time values of specific event counters
- Provide performance indications
 - Extra cycles taken to execute instructions
 - May be due to memory contentions (Flash waitstates)
 - Cycles of overhead caused by handling exceptions
 - Cycles spent in sleep mode
 - Number of cycles spent performing memory accesses
 - Number of folded branch instructions

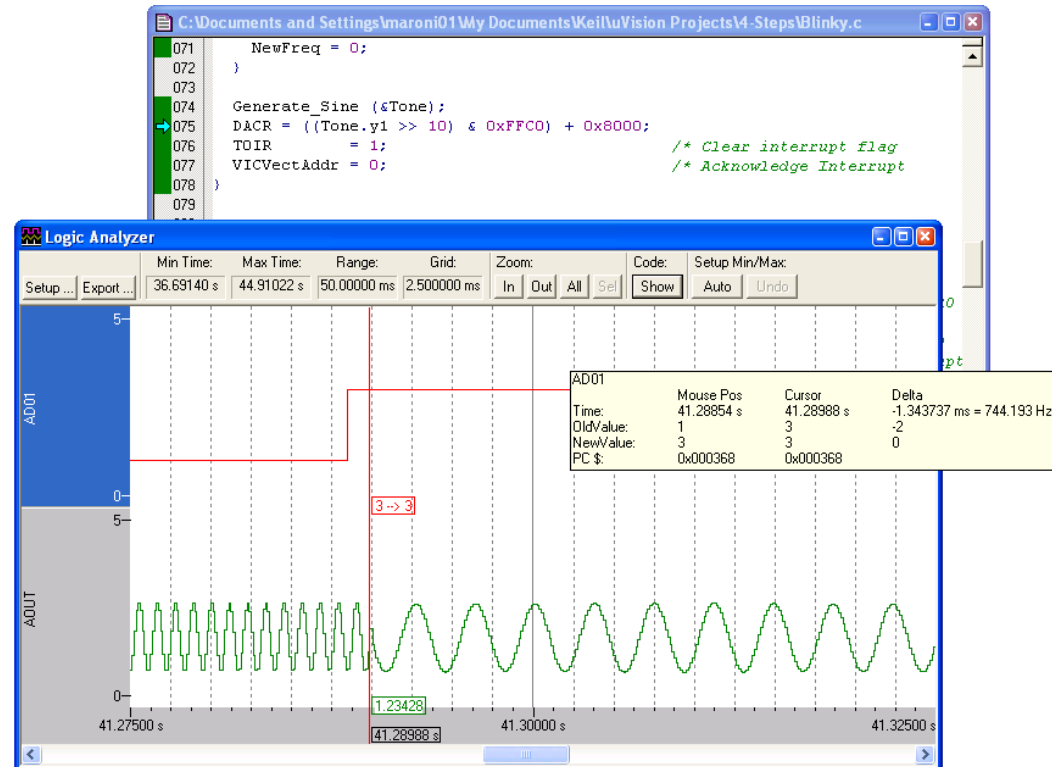


The screenshot shows a window titled "Event Counters" with a list of performance metrics. Each metric is displayed in a table-like format with a label, a numerical value, a small reset button (0), and a description.

Counters			
CPICNT:	2987625	0	Extra Cycles per Instruction
EXCCNT:	818	0	Exception overhead cycles
SLEEPcnt:	2048	0	Sleep Cycles
LSUCNT:	58956	0	Load Store Unit Cycles
FOLDcnt:	46	0	Folded Instructions

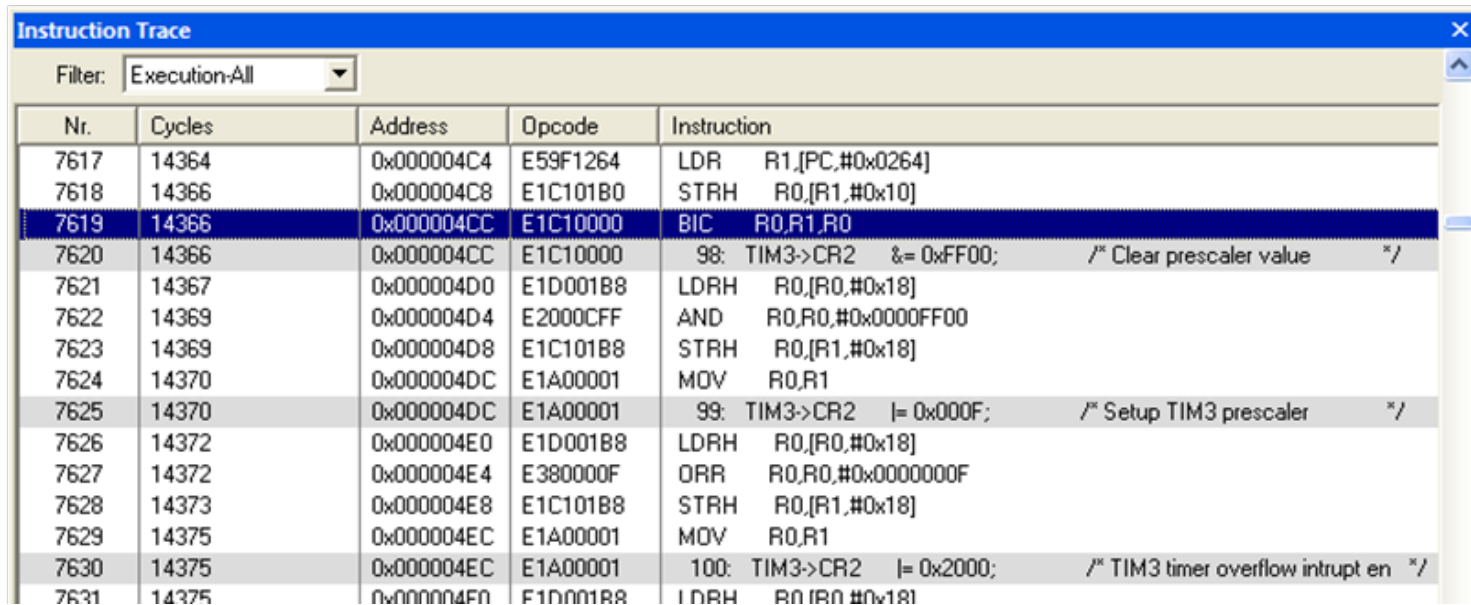
Logic Analyzer (DWT)

- Allows signals to be monitored graphically
 - Monitor variables in the application
- Accurate timing
 - Easy, fast analysis of signal timing with access to source code
 - View delta changes from cursor to current location
- Code analysis
 - View instruction that caused variable change



Instruction Trace (ETM)

- Execution history of all executed instructions
 - Instruction Trace window displays: cycle count (timing) and assembly code synchronized to the C source code.




Nr.	Cycles	Address	Opcode	Instruction
7617	14364	0x000004C4	E59F1264	LDR R1,[PC,#0x0264]
7618	14366	0x000004C8	E1C101B0	STRH R0,[R1,#0x10]
7619	14366	0x000004CC	E1C10000	BIC R0,R1,R0
7620	14366	0x000004CC	E1C10000	98: TIM3->CR2 &= 0xFF00; /* Clear prescaler value */
7621	14367	0x000004D0	E1D001B8	LDRH R0,[R0,#0x18]
7622	14369	0x000004D4	E2000CFF	AND R0,R0,#0x0000FF00
7623	14369	0x000004D8	E1C101B8	STRH R0,[R1,#0x18]
7624	14370	0x000004DC	E1A00001	MOV R0,R1
7625	14370	0x000004DC	E1A00001	99: TIM3->CR2 = 0x000F; /* Setup TIM3 prescaler */
7626	14372	0x000004E0	E1D001B8	LDRH R0,[R0,#0x18]
7627	14372	0x000004E4	E380000F	ORR R0,R0,#0x0000000F
7628	14373	0x000004E8	E1C101B8	STRH R0,[R1,#0x18]
7629	14375	0x000004EC	E1A00001	MOV R0,R1
7630	14375	0x000004EC	E1A00001	100: TIM3->CR2 = 0x2000; /* TIM3 timer overflow interrupt en */
7631	14375	0x000004F0	F1D001B8	LDRH R0,[R0,#0x18]

- Instruction Trace is useful to analyze sporadic problems
 - Data corruption by incorrect interrupt/thread protection
 - Incorrect timing caused by interrupt/thread nesting

Sporadic Data Problem (ETM)

```
void Alert (void) {                                // Alarm Function
    if (clock.min != 59 &&                          // Validate Time
        clock.hour != 12) {
        debug_printf ("System Should never be there");
    }
}

void CheckAlert (void) {                          // check for alarm at 12:59
    if (clock.min == 59 &&                          // check minute for 59
        clock.hour == 12) {                      // check hour for 12
        Alert ();                                // call Alarm Function
    }
}
```



Instruction Trace

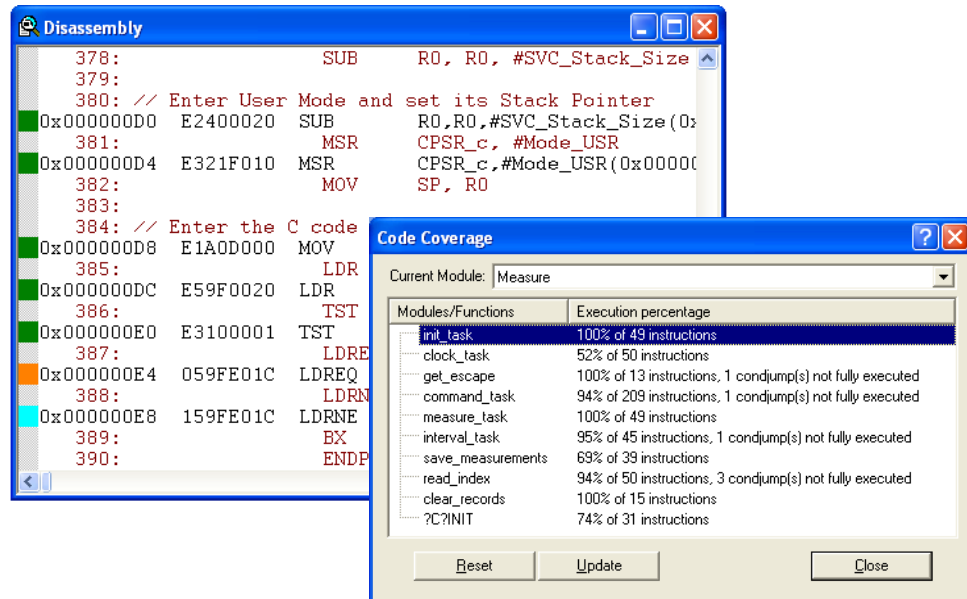
Filter: Execution-All

Nr.	Cycles	Address	Opcode	Instruction
7617	14364	0x000004C4	E59F1264	LDR R1,[PC,#0x0264]
7618	14366	0x000004C8	E1C10180	STRH R0,[R1,#0x10]
7619	14366	0x000004CC	E1C10000	BIC R0,R1,R0
7620	14366	0x000004CC	E1C10000	98: TIM3->CR2 &= 0xFF00; /* Clear prescaler value */
7621	14367	0x000004D0	E1D001B8	LDRH R0,[R0,#0x18]
7622	14369	0x000004D4	E2000CFF	AND R0,R0,#0x0000FF00
7623	14369	0x000004D8	E1C10188	STRH R0,[R1,#0x18]
7624	14370	0x000004DC	E1A00001	99: TIM3->CR2 = 0x000F; /* Setup TIM3 prescaler */
7625	14370	0x000004E0	E1D001B8	LDRH R0,[R0,#0x18]
7626	14372	0x000004E4	E2000CFF	AND R0,R0,#0x0000FF00
7627	14372	0x000004E8	E1C10188	STRH R0,[R1,#0x18]

Instruction Trace shows Interrupt Execution within the compare statement

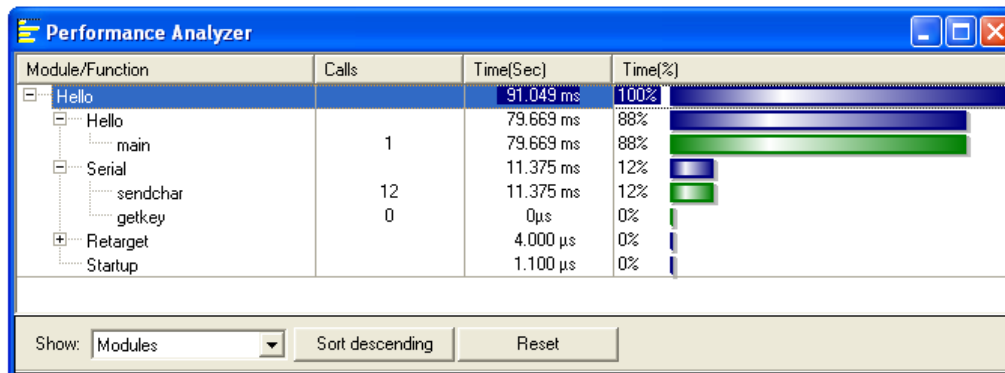
Code Coverage (ETM)

- Complete software validation requires code coverage
 - Product liability and industry standards (IEC61508) demand for testing according to “State of Science and Technology” methodologies
- ETM enabled devices provide complete instruction stream
 - Non-intrusive - use final, optimized code at full speed
- Feedback provided directly in the debugger window
 - Source & disassembly view
 - Color-coded details for individual instructions
 - Summary analysis by function or module
- Log File Support
 - Coverage information can be saved for documentation



Execution Time Profiling and Analysis (ETM)

- Instruction Trace provides timing information
 - Identify where most time is spent in your application



- Isolate problems by finding which C statements take longer than expected to execute

The C:\temp\WC4_Clip\clip.c window shows the following code with execution times:

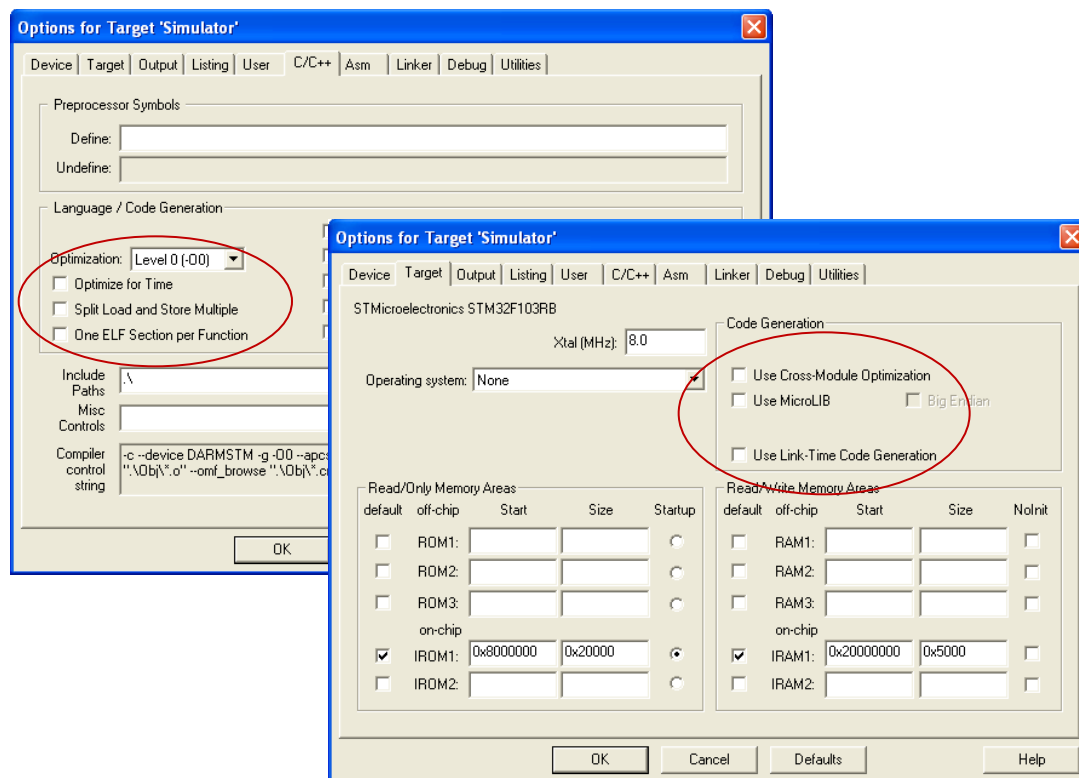
```
131 2.628 ms void DPhase (short s, struct quad *b) {
132      int res;
133 26.942 ms res = B0*s + B1*b->x[0] + B2*b->x[1];
134
135 1.314 ms b->x[1] = b->x[0];
136 1.314 ms b->x[0] = s;
137 1.314 ms b->y[1] = b->y[0];
138 1.971 ms b->y[0] = (res >> 16);
139 1.971 ms }
140
141 struct avarage {
```

The Disassembly window shows the assembly instructions for the DPhase function:

```
131: void DPhase (short s, struct quad
132:   int res;
2.628 ms 0x0000029A B470 PUSH {R4-R6}
133:   res = B0*s + B1*b->x[0] + B2*b->
134:
1.971 ms 0x0000029C 4C55 LDR R4,[PC,#0:
1.971 ms 0x0000029E 4B56 LDR R3,[PC,#0:
1.971 ms 0x000002A0 680A LDR R2,[R1,#0:
1.971 ms 0x000002A2 4344 MUL R4,R0
1.971 ms 0x000002A4 4353 MUL R3,R2
```

Compiler Optimizations

- Compilation tools feature range of optimization options
 - These enable you to target your application to its needs
- User control
 - Select optimization options to suit target application/device
- Intelligent tools
 - Automatic optimizations, no changes required in source code
 - Does not interfere with the functional behaviour of the code



Optimization examples - RVCT

- Application optimized for smallest code size †

Optimization	Compile Size	Size Reduction	%age Reduction
No optimization	13,656 Bytes		
MicroLIB	8,960 Bytes	4,696 Bytes	34.4%
Optimization Level 2	12,936 Bytes	720 Bytes	5.3%
Both Options	8,264 Bytes	5,392 Bytes	39.5%

- Application optimized for best performance ‡

Measurement	No Optimization	Fully Optimized	Improvement
dhry_1	2.829s	1.695s	40.1% faster
dhry_2	2.014s	1.011s	49.8% faster
µs per Dhrystone	138.0	70	49.3% faster

- C:\Keil\ARM\Boards\Keil\MCBSTM32\Measure\Measure.uv2 †
- C:\Keil\ARM\Examples\DHRV\DHRV.uv2 using an Atmel AT91SAM7 device ‡

END

Thank you