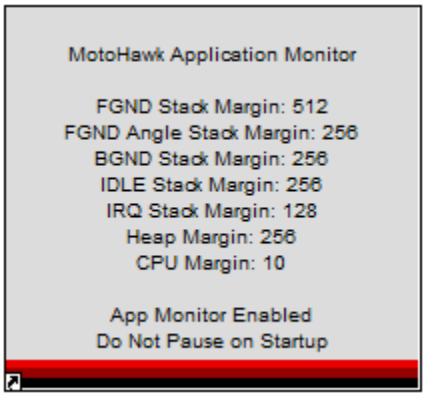


Debugging A MotoHawk Application using the Application Monitor

Author(s):

New Eagle Consulting
3588 Plymouth Road, #274
Ann Arbor, MI 48105-2603
Phone: +1 (734) 929-4557

Ben Hoffman
Product Development Lead /
Senior Software Engineer
bhoffman@neweagle.net

Table of Contents

1. Debugging Embedded Control Modules Background

The MotoHawk control system application development tool chain allows for rapid development of control algorithms and produces build outputs that can be seamlessly deployed to rugged, production-validated control modules (ECUs). This system generally works great, however, every once in a while, you are reminded that you are developing in a constrained embedded environment. Inherent in this environment are limitations on CPU throughput and memory resources. Additionally, creating software that is deployed to sealed, production ready hardware limits the ability to utilize debugging tools common in the embedded software development space, such as a JTAG debugger. Such tools allow the developer to halt CPU execution and inspect CPU state, and even step though the software source code (C files) looking for defects during execution. Thankfully, with a robust, mature code-generation platform like MotoHawk the types of issues that these tools are generally used to solve have been resolved already and are available in proven software blocks for you to use in your application. In reality, for the controls developer in the MotoHawk context the source code is really the Simulink model and inspecting the derived (generated) C source code would not necessarily be of much value. However, due to the nature of the inherent constraints of embedded hardware, (limited processing power, limited memory, etc…) at some point something such as a stack overflow, or processor watchdog will happen as you approach these limits. In the general, this would normally cause the ECU to exhibit continual or spurious processor reset or other strange behavior. As a general rule, when the processor resets there is not much indication as to the cause of the error because the state of the processor is wiped on reset. Without additional tools, tracing the error can be quite difficult. This is where the MotoHawk Application Monitor is meant to help.

2. Application Monitor Basics

The Application Monitor performs continuous run-time checks on an executing application, monitoring key metrics and will halt the application if something goes wrong. When it stops the application it writes an error status to some user accessible variables for inspection and leaves the calibration protocol running so you can connect to the ECU and inspect these messages to determine the issue. By halting the application before the module resets, the error state can be captured by the user using a calibration tool like MotoTune.

3. Terminology

- Stack – In the discussion below the term ‘stack’ is used to refer to an area of RAM set aside for storing state for local usage or to recover at a later time. For instance, when the code enters a function call, most local variables are stored on the stack and used for calculations during that function. If, inside the function, another function is called then the processor state (registers, return address, etc..), called a ‘stack frame’, needs to be pushed onto the stack for recall when the called function returns. In MotoHawk, stacks are of predetermined size and hence

exceeding the allocation will generally cause issues as it may attempt to overwrite other allocated RAM.

- Heap – In the discussion below the term 'heap' refers to an area of RAM set aside for storing memory that is allocated at runtime. For instance, the OS framework utilizes heap storage for communication queues and resource instantiation. Applications that have a lot of communications and/or resource (I/O) usage will use more heap.

4. Application Monitor Setup

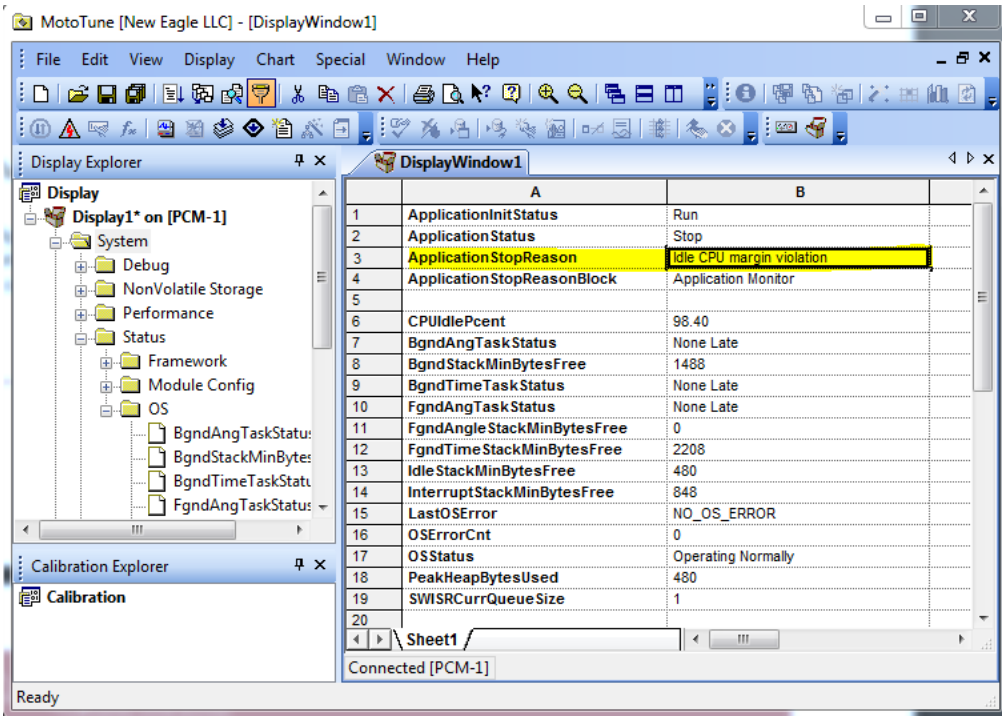
To set up the Application Monitor, you must add it to your model. The 'Application Monitor' block can be found in the Simulink Library browser at 'MotoHawk->System Debug Blocks'. There are two versions, the 'motohawk_app_monitor' for 'legacy' modules (5xx based) and the 'Application Monitor (2nd Gen)' for 'MotoCoder' modules, (S12, 55xx, and newer etc..). After adding it to your application you may wish to configure its settings. Although the defaults usually work fine to get started (except on some of the smaller modules), you may want to work with each of the following parameters to tailor the configuration to your needs:

- Foreground Stack Margin [bytes]
  - Configures the required headroom (margin) for the Foreground Task(s) stack under which the Application Monitor will halt the application.
- Foreground Angle Stack Margin [bytes]
  - Configures the required headroom (margin) for the Foreground Angle Task(s) stack under which the Application Monitor will halt the application. This parameter relates to tasks which take place based on angle triggers.
- Background Stack Margin [bytes]
  - Configures the required headroom (margin) for the Background Task(s) stack under which the Application Monitor will halt the application.
- Idle Stack Margin [bytes]
  - Configures the required headroom (margin) for the Idle Task stack under which the Application Monitor will halt the application. Note that in most cases the calibration protocol will run in the idle task.
- Interrupt Stack Margin [bytes]
  - Configures the required headroom (margin) for the Interrupt Task(s) stack under which the Application Monitor will halt the application.
- Heap Margin [bytes]
  - Configures the required headroom (margin) for the heap usage. If the peak heap usage comes within this margin of the amount of allocated heap memory the Application Monitor will halt the application. More on this in 'Section 6 – Tuning the Application Monitor'.
- CPU Margin [%]
  - Configures the minimum amount of idle CPU capacity that must be maintained. If the CPU idle capacity goes below this value (CPU is too busy due to large tasks, etc…), the Application Monitor will halt the application.

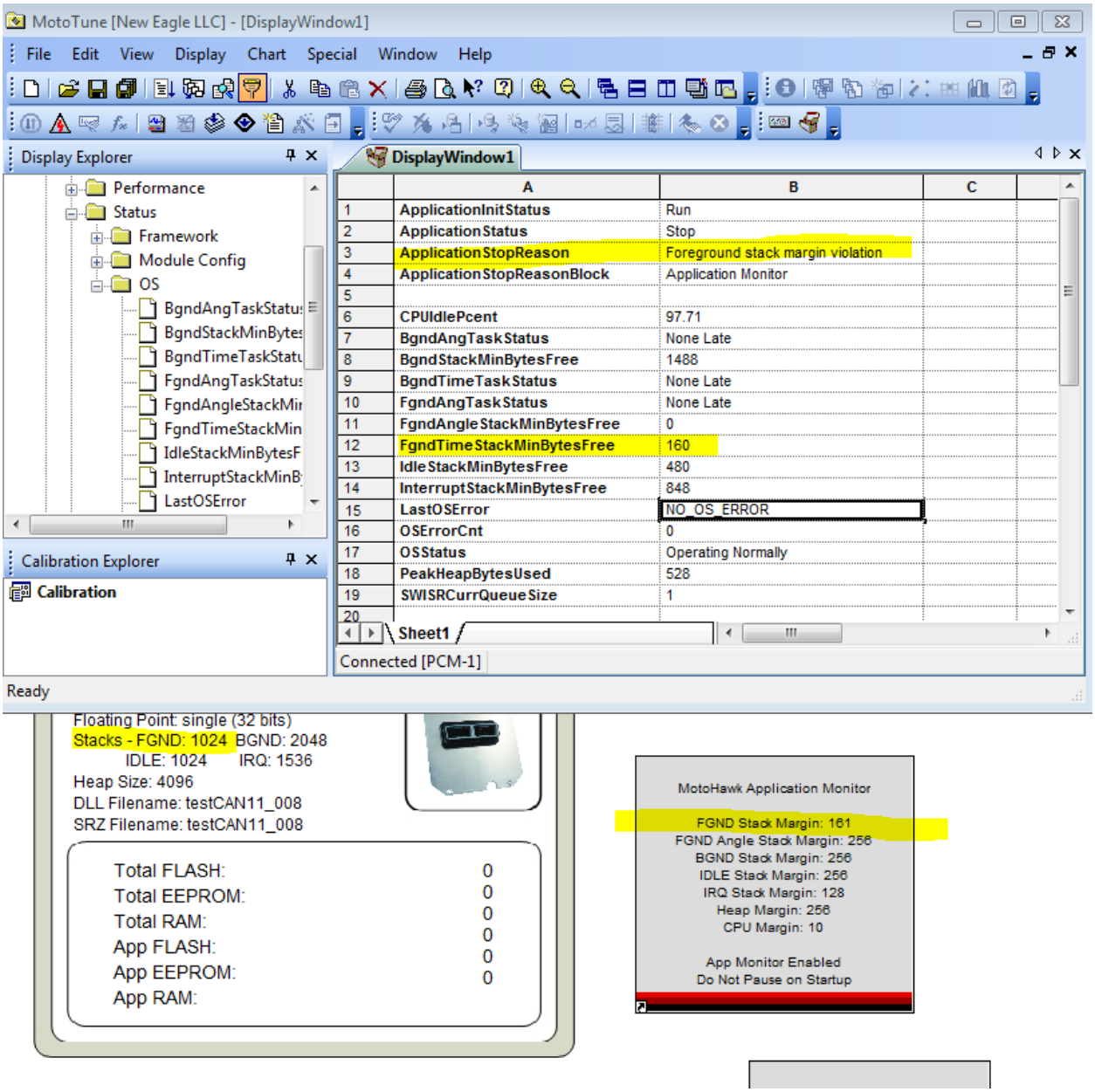
- On Startup – This allows you to configure the startup behavior of the application for debugging. The Application Monitor can pause execution of the application to allow you to connect with the calibration tool to inspect the state of the controller before the subsequent steps are executed.
    - Do No Pause on Startup [Default]
        - Application executes normal startup running user tasks as typically expected. (Normal Operation)
    - Pause Before Model Initialization
        - The application will halt before executing model initialization code.
    - Pause After Model Init – Before STARTUP Event
        - Model-generated initialization code is executed. The model initialization code takes care of setting default values for data stores and other global variables, as well as initial time values for dT blocks and initial values on 1/z blocks.
    - Pause After STARTUP Event – Before Run
        - The application pauses after the STARTUP event (Operating System Event). Any code you have placed in STARTUP event triggered subsystems will have been executed.

  Each of these options allows you to effect the startup behavior of the ECU under control of the Application Monitor. This can be useful in debugging situations where the software you have flashed into the controller fails to operate.

5. Examples



In the screenshot above we can see that the Application Monitor has stopped the application for 'Idle CPU margin violation'. This occurred because the CPUIdlePcent value dropped below the CPU Margin [%] specified in the Application Monitor block. To resolve this you would normally either slow down the Foreground Rate (10ms vs 5ms), optimize your code, or partition functionality among different tasks depending on your situation.



In the example above, the Application Monitor has halted the application due to 'Foreground stack margin violation'. Although this is a contrived example (your settings would generally be larger) you can see the Foreground Stack Margin [bytes] setting in the Application Monitor is set to 161 bytes. The Target Definition block has a Foreground Stack size set to 1024 bytes. In MotoTune you can see the FgndTimeStackMinBytesFree is 160 which is below the margin threshold setting for the Application Monitor, hence it has halted the application. To resolve this you would normally just allocate more Foreground stack space in the Target Definition block.

6. Tuning the Application Monitor

The Application Monitor may halt your application for the reasons below. The reason will be indicated in 'System->Debug->ApplicationStopReason'. Some information on tuning for each case is described.

- Requested by user
  - User requested via 'System->Debug->ApplicationStatus'. This is a manual halt caused by user intervention. The 'ApplicationStatus' variable can also be used to restart the application.
- Foreground stack margin violation
  - Monitoring FgndTimeStackMinBytesFree in 'System->Status->OS' shows the cumulative minimum free bytes of allocated stack for Foreground Task(s). If this value dips below the established margin, the Application Monitor will halt your application. You can adjust Foreground Stack Margin [bytes] in the Application Monitor block, or increase the Foreground Stack in the Target Definition Block.
- Background stack margin violation
  - Monitoring BgndStackMinBytesFree in 'System->Status->OS' shows the cumulative minimum free bytes of allocated stack for Background Task(s). If this value dips below the established margin, the Application Monitor will halt your application. You can adjust Background Stack Margin [bytes] in the Application Monitor block, or increase the Background Stack in the Target Definition Block.
- Idle stack margin violation
  - Monitoring IdleStackMinBytesFree in 'System->Status->OS' shows the cumulative minimum free bytes of allocated stack for the Idle Task. If this value dips below the established margin, the Application Monitor will halt your application. You can adjust Idle Stack Margin [bytes] in the Application Monitor block, or increase the Idle Stack in the Target Definition Block. In most scenarios the calibration protocol is running in the idle task.
- Interrupt stack margin violation
  - Monitoring InterruptStackMinBytesFree in 'System->Status->OS' shows the cumulative minimum free bytes of allocated stack for Interupt Task(s). If this value dips below the established margin, the Application Monitor will halt your application. You can adjust Interrupt Stack Margin [bytes] in the Application Monitor block, or increase the Interrupt Stack in the Target Definition Block.
- Heap margin violation
  - Monitoring PeakHeapBytesUsed in 'System->Status->OS' shows the cumulative peak usage of heap. If this value exceeds the heap allocation in the target definition block MINUS the Heap Margin [bytes] set in the Application Monitor block, the Application Monitor will halt your application. You can adjust Heap Margin [bytes] (smaller) or increase the heap allocation in the Target Definition Block.
- Idle CPU margin violation

  - Monitor using 'System->Performance->CPUIdlePcent' which shows the amount of CPU time that is not in use by your application's tasks. You can reduce the allowed margin (less CPU idle headroom), optimize your task(s) code, or reallocate functionality within tasks. For instance, if you are exceeding the CPU throughput due to a large FGND_RTI_PERIODIC task, you may look to move some functions to a slower task such as FGND_2XRTI_PERIODIC, FGND_5XRTI_PERIODIC or even into background tasks such as BGND_BASE_PERIODIC and BGND_BASEx2_PERIODIC. If reorganizing your application is not possible, you might use a MotoHawk Trigger Definition to slow the task down.
    - Further debugging task timing can be done using the following variables from 'System->Performance':
      - CPUTime_AppISR
      - CPUTime_Bgnd
      - CPUTime_CAN
      - CPUTime_FgndAngle
      - CPUTime_FgndTime
      - CPUTime_MIOS
      - CPUTime_Serial
      - CPUTime_TPU
      - CPUTime_TimerISR
- OS error detected
  - This occurs when an operating system error has been detected. For further debugging you should inspect 'System->Status->OS->LastOSError'. The values for this variable will vary with the target ECU, but may include:
    - PROSAK_TPU_ALLOCATE
    - PROSAK_DEADLOCK_PROT
    - QSPI_SEND_SIZE_ZERO
    - QSPI_SEND_SIZE_TOO_BIG
    - QSPI_RD_SIZE_TOO_BIG
    - QSPI_RD_SIZE_ZERO
    - QSPI_WR_SIZE_TOO_BIG
    - QSPI_WR_SIZE_ZERO
    - QSPI_CMD_SIZE_TOO_BIG
    - QSPI_CMD_SIZE_ZERO
    - IGN_DIAG_MDA_ALLOCATE
    - EEDataDirSizeMisMatch
    - EE_WRITE_QUEUE_OVFLOW
    - ESC_HBRIDGE_MDA_ALLOCATE
    - ETC_DIAG_MDA_ALLOCATE
    - DFLT_ENCODER_CREATE_ERR
    - ADC_TPU_ALLOCATE
    - SWISR_QUEUE_OVERFLOW
    - WATCHDOG_RESET
    - CHECKSTOP_RESET

- SPIEE_WR_STAT_REG_ERR
- SPIEE_QUEUE_OVFL_STATUS
- SPIEE_WRITE_FAIL_ERR
- Requested TPU Func not in mask
- Unable to allocate QSPI knock
- Stack has overflowed!
- TPU Limits not setup
- INVALID TPU MASK
- TPU Mask Corrupt
- Sequence Create with No Encoder
- Illegal Sequence ISR
- TPU Mask Version is insufficient
- MUX PSP requested for 2-Stroke
- Ext Flash Acc Denied
- Ext Flash Erase Fail
- Ext Flash Write Fail
- Ext Flash Allign Error
- EXT_FLASH_WR_ZERO_SIZE
- Illegal Ext Flash Addr
- Error: Write to non-existent Flash
- Too many Logs want to be active
- Bank Seq Destroy Problem
- Transient Seq Destroy Problem
- Transient Seq Update Problem
- SPIEE System had a memory problem
- Spent too long waiting for a sync return
- TPUC Mask does not pair with TPUA/B mask
- The Module Configuration was not available
- Diagnostic Mux did not create
- FGND Angle event queue overflow
- FGND Time event queue overflow
- EEPROM write to CAMDelay failed
- SleepTask call from in ISR
- Bad Kernel API call detected
- Kernel Internal
- SPIEE Device create
- NAND Flash timeout on Write
- NAND Flash timeout on Erase
- NAND Flash WP
- NAND Flash RDY/BUSYb
- NAND Flash Access
- NAND Flash Device
- USB Core Stack Initialisation

- USB Stack Assert
- EXT_EE_ACCESS
- VARCAM_PATTERN_SETUP
- FP Divide by Zero
- Unable to allocate requested memory on heap
- Unable create Fgnd Angle task
- FP Denormalized number
- RPMVector Buffers Full
- RPMVector Overrun
- Failed to Write TPU Self Modify Parameter

- Further debugging of these errors will depend on the error and may require further support from New Eagle. Actual error list will vary with ECU target.

7. Establishing Validation Confidence using the Application Monitor

Another use for the Application Monitor is to establish a safety margin for key operating parameters. Once established, these can be verified during system testing and validation. As an example, if your system requires a safety margin of 30% CPU utilization (e.g. system runs <70% CPU usage at all times) then you can set that in the application monitor and conduct your validation testing. After testing you have confidence that, for the areas of code executed as part of your test plan, you will not exceed the safety margin. You have the option of disabling the Application Monitor via calibration for production releases.

8. Summary

The Application Monitor provides a tool for investigating critical control software issues that would otherwise be very difficult to resolve. By properly configuring the Application Monitor you can avoid trial and error troubleshooting when things go wrong during application development and system testing.