Static Analysis Considerations for Stack Usage
ESC-423

Gerald T. Rigdon
Fellow, Software Engineering
Boston Scientific, Inc.
gerald.rigdon@bsci.com

Hiten Doshi
Fellow, Software Engineering
Boston Scientific, Inc.
hiten.doshi@bsci.com

Xin Zheng
Software Engineer
Boston Scientific, Inc.
xin.zheng@bsci.com

In [3], David N. Kleidermacher, chief technology officer at Green Hills Software, discussed future directions for static analysis. He identified that using static analyzers for detecting stack overflow conditions is a prime example of how having these capabilities in tools could be beneficial, especially in safety critical applications.

One particular tool that offers a solution is StackX from Express Logic, Inc. On their website [4], they state the following:

"StackX™ is a unique, patent pending, development tool that helps developers avoid stack overflow problems that traditionally have plagued embedded systems. StackX performs a comprehensive analysis of a complete application at the executable code level, computing the worst-case stack usage the application can experience. Express Logic is the only RTOS company that can identify how large a stack customers need for each application thread and automatically alerts them if they fail to allocate sufficient memory!"

The website [4] goes on to discuss how the decision to allocate memory for stack use has typically been a trial and error process and then quotes Jack Ganssle [2], who has been a long time contributor to Embedded Systems Magazine, as saying:

"*With experience, one learns the standard, scientific way to compute the proper size for a stack: Pick a size at random and hope."*

Although it is not the goal of this paper to discuss the various COTS (Commercial Off The Shelf) solutions for detecting stack overflow, we did take the liberty of inquiring about StackX for use with our firmware, but the tool did not support our build environment.

In the absence of COTS tools for detecting stack overflow, the embedded community has embraced effective alternative methods. Let us now explore one such method and then consider a customized approach.

Using Common and Customized Solutions

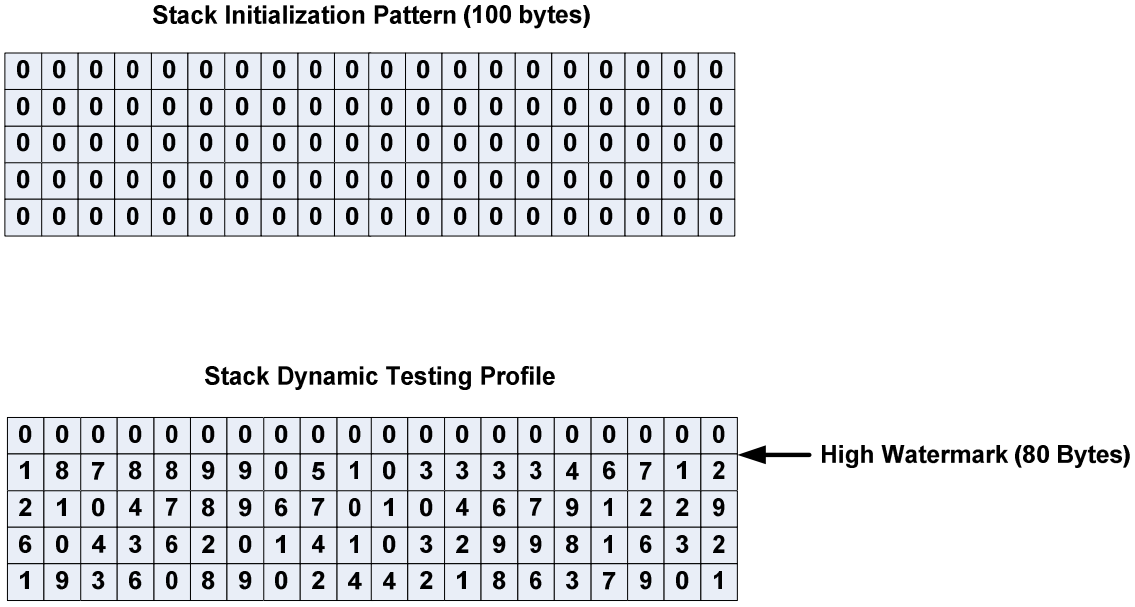
In [1], we discussed a project, namely CSAP (Custom Static Analysis Project), between Boston Scientific, Inc. and Grammatech, Inc. The driving objective of that project was to

customize CodeSonar, Grammatech's flagship static analysis tool, to meet a very specific set of requirements based on our domain. The primary reason we undertook this customization effort was to tackle the analysis of shared data (AKA race conditions) in our firmware. However, as described in [1], there were other project deliverables, including a CFUA (Call Frame Usage Analysis), in particular, as a customized solution to detecting stack overflow. Since [1] only referenced CFUA at a very high level, we will take the opportunity to elaborate on that particular analysis for the purpose of this paper.

Before we get into the details of CFUA, it is worth exploring a common method used in many embedded environments. This method uses a watermark approach to determine stack usage during dynamic testing. To illustrate, we present an arbitrarily chosen stack size of 100 bytes, as shown in Figure 1. For the sake of simplicity, assume all stack memory is initialized to zero (although developers may prefer a specific pattern for initialization). This effectively establishes a known initialization state, which allows one to run dynamic test scenarios likely to execute almost all branches of execution in order to determine stack usage.

This approach allows one to generate a profile of stack usage since the stack memory can be inspected after testing to determine the high watermark. In the example in Figure 1, the highest watermark indicates a usage of 80 bytes, or 80% of the allocated stack, during testing.

Figure 1: Watermark Approach to Stack Analysis



Although this is an effective method employed in many embedded environments, it is only as good as the dynamic testing scenarios. Therefore, this approach provides an accurate estimate only if dynamic testing actually covered the paths of deepest stack penetration. In complex systems, the watermark approach would be associated with the risk of representing a lower bound that is likely less than the actual stack usage.

However, if we could also establish an upper bound for worst-case stack usage, then the actual stack usage would essentially be captured in the following expression:

watermark stack usage $\leq$ actual stack usage $\leq$ worst-case stack usage

Let us now consider the details of our customized solution to calculating the worst-case stack usage.

Customized Stack Analysis Tooling

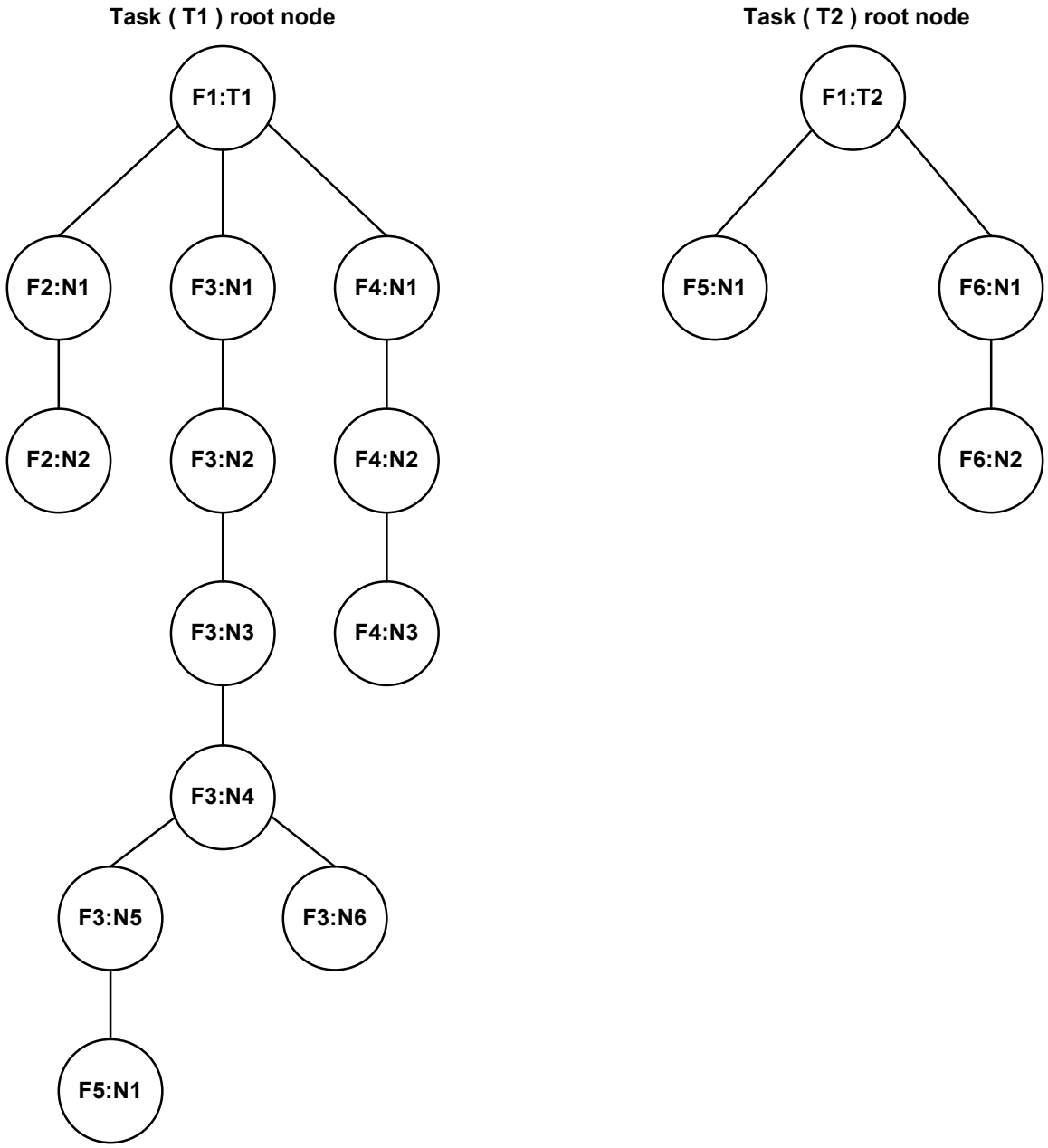
As described in [1], our medical device firmware is a preemptive multi-threaded design implemented in the C language. The customization involved the creation of an input task list file to CodeSonar that identified all of our task threads and their associated priority levels. In addition, the effort also involved the creation of an input function specifier file, which consists of a comprehensive list (CSV file) of all file:function pairs in the firmware and tagging these with various attributes, such as the call frame size (stack usage) for each pair. [1] The other attributes capture information about the file:function pairs based on our product configuration model. More specifically, we have a common source code base for all product models, but a given build uses only the firmware components applicable to generating a binary image for that particular product.

With these inputs, a general approach to calculating stack usage begins to emerge. By leveraging the capability of CodeSonar to parse the firmware and create call trees, we were able to use our input files to great advantage. Each task thread entry point indicated in the input task list file is a root node, and each call tree built from that root node consist of branches that represent all paths of execution. Each individual branch from the root node to the terminating node is represented by a list of file:function nodes associated with said descriptive attribute inputs from the input function specifier file. Therefore, information about the call frame size for each node is sufficient to enable a stack calculation for the entire call tree and each branch of execution. Using a Depth First Search algorithm approach, the branch(es) in the call tree using the most stack would represent the worst-case stack usage for the task thread associated with root node of the call tree. Finally, the worst-case stack usage for the entire product software is calculated by summing the maxima of worst-case stack usage at each priority level.

To visualize this, we have provided Figures 2, 3, and 4, which illustrate this customized solution by breaking down the problem into multiple steps. For the first step, assume that we only specify the task threads (root nodes) as inputs in the input task list file. In this case, we have specified an input file named “F1” that has two task threads named “T1” and “T2.” Also, assume that the other files and functions in the call tree represent the remaining source code for our example, which was used to create the watermark profile shown in Figure 1. Given this configuration, Figure 2 illustrates what the call tree would look like:

[1] We use a Perl script to parse all assembly language output for our source code to determine stack usage for each file:function pair, per our compiler tool vendor’s specifications.

Figure 2: Call Tree



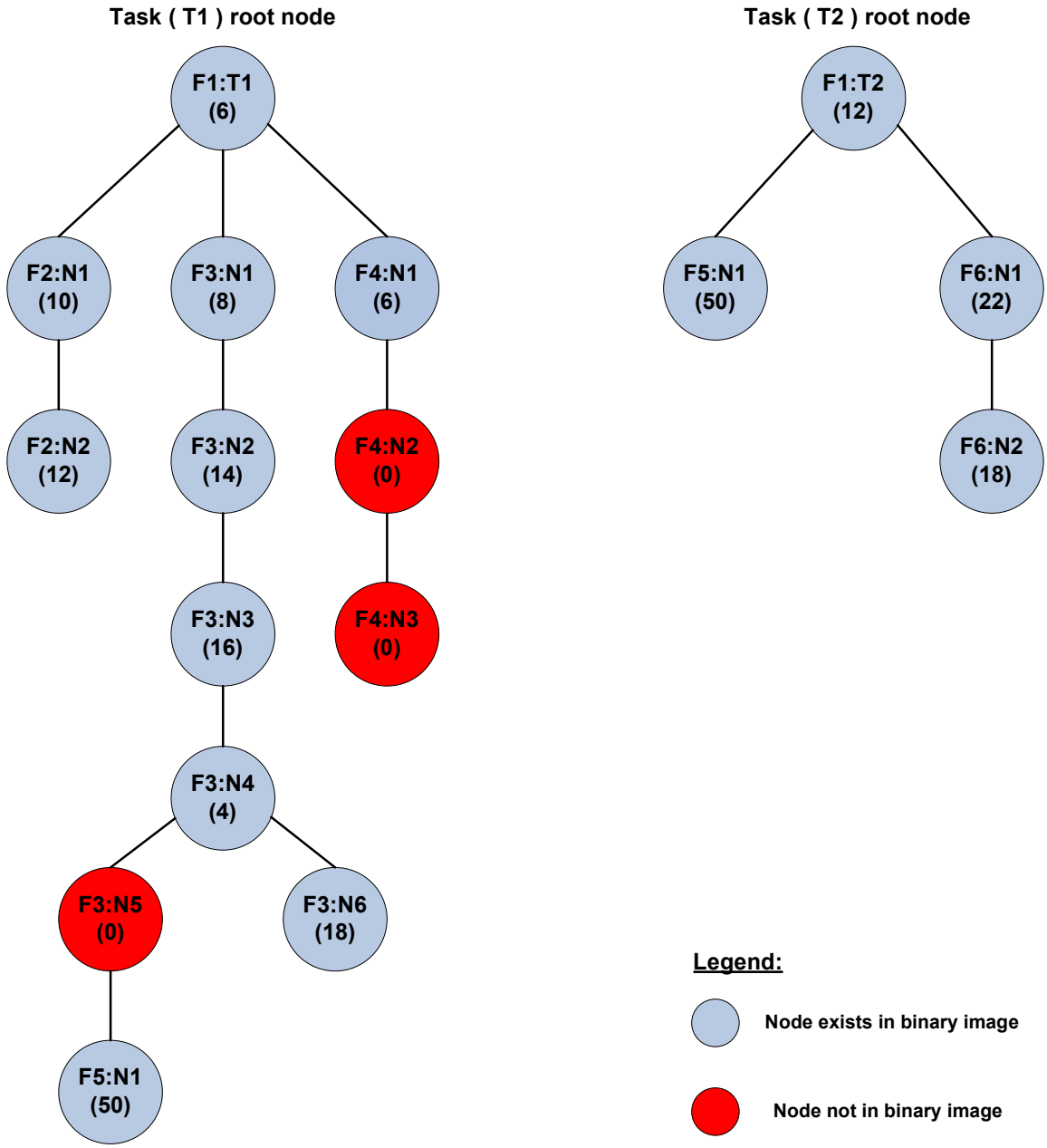
For the next step, consider a set of inputs with attributes that further describe the file:function pairs in the source code, which are the nodes in the call tree. For each node, we specify the call frame size, the color, and a binary image attribute. Earlier, we referred to the fact that our firmware has a common source code base, but a given build uses only the firmware components applicable to generating a binary image for a particular product model. The input specification shown in Figure 3 supports this build environment. In this example, F3:N5, F4:N2, and F4:N3 are not part of the product configuration and are excluded from the binary image. These file:function pairs (nodes) are designated by the color "RED," and since they do not use any stack resources, their call frame sizes are set to "0."

Figure 3: Function Specifier

"C" File	FunctionName	Call Frame Size Bytes	Color	In Binary Image
F1	T1	6	BLUE	Yes
F1	T2	12	BLUE	Yes
F2	N1	10	BLUE	Yes
F2	N2	12	BLUE	Yes
F3	N1	8	BLUE	Yes
F3	N2	14	BLUE	Yes
F3	N3	16	BLUE	Yes
F3	N4	4	BLUE	Yes
F3	N5	0	RED	No
F3	N6	18	BLUE	Yes
F4	N1	6	BLUE	Yes
F4	N2	0	RED	No
F4	N3	0	RED	No
F5	N1	50	BLUE	Yes
F6	N1	22	BLUE	Yes
F6	N2	18	BLUE	Yes

With these additional inputs to the static analysis tool, the call tree effectively looks like Figure 4.

Figure 4: Call Tree with Stack Usage



Based on the call tree in Figure 4, the stack used for each path would be as follows:

Task (T1) root node:

[Path1]➔ F1:T1(6) + F2:N1(10) + F2:N2(12) = 28 bytes

[Path2]➔ F1:T1(6) + F3:N1(8) + F3:N2(14) + F3:N3(16) + F3:N4(4) + F3:N5(0) + F5:N1(0) = 48 bytes

[Path3]➔ F1:T1(6) + F3:N1(8) + F3:N2(14) + F3:N3(16) + F3:N4(4) + F3:N6(18) = 66 bytes

[Path4]➔ F1:T1(6) + F4:N1(6) + F4:N2(0) + F4:N3(0) = 12 bytes

Task (T2) root node:

[Path1]➔ F1:T2(12) + F5:N1(50)= 62 bytes

[Path2]➔ F1:T2(12) + F6:N1(22) + F6:N2(18) = 52 bytes

Before we proceed, let us revisit root node (T1) [Path2], which resulted in "48" bytes of stack usage. It is apparent that the call frame size used for F5:N1 in the calculation was "0" while the Function Specifier table in Figure 3 indicated a value of "50." This is because the tool is aware that an unreachable function (in the context of this call tree) does not consume stack resource. While the F5:N1 node is in the binary image, it cannot be executed in this path because its caller, F3:N5, is not in the binary image. However, in root node (T2) [Path1], F5:N1 is reachable since its caller, F1:T2, is in the binary image. Therefore, the call frame size used is "50," as expected. To summarize, when the algorithm encounters a node that is not in the binary image, all its child nodes' call frame sizes become "0" for that path.

For the final step in calculating worst-case stack usage for the entire product software, we will assume each task thread is associated with a different task priority level. If we assume root node (T1) is a Level1 priority task and root node (T2) is a Level2 priority task, given a simple preemptive design for our example, the calculation becomes:

Max(Level1) + Max(Level2) = worst-case stack usage

(T1) [Path3] + (T2) [Path1] Or 66 + 62 = 128 bytes

By combining the watermark profile shown in Figure 1 with the worst-case usage calculation, we are able to bound the problem as follows:

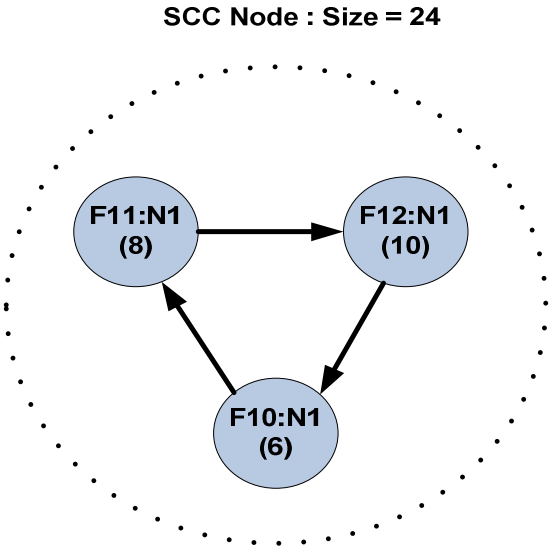
80 bytes (watermark) ≤ actual stack usage ≤ 128 bytes (static analysis)

One possible scenario during dynamic testing resulting in this particular watermark value could be that task thread (T1) was preempted by task thread (T2) during execution of F2:N2 and proceeded to execute [Path2] for T2. The stack penetration during execution of F2:N2 would have been 28 bytes, and [Path2] for T2 would add 52 bytes more after preemption. In any case, the worst-case analysis indicated stack penetration that exceeds the stack size of 100 bytes we allocated in this example. Given these results, a decision will need to be made as to whether all paths are viable and whether the worst-case should be accommodated.

Recursive Considerations

Now that we have a good understanding of our general approach, let us dig down further and discuss the implications of recursive components. Another project deliverable we had, as described in [1], was to identify all SCC (Strongly Connected Components) using Tarjan's algorithm. Figure 5 shows an example of SCCs consisting of three nodes, with call frame sizes 6, 8, and 10.

Figure 5: SCC



In terms of stack usage, the entire SCC group becomes an abstract node representing its sub-components. When the algorithm encounters a node that is part of an SCC group, it will use the collective call frame size of the abstract SCC node. Thus, it assumes each node is visited only once. This is not the case in reality, if we assume a good design that does not allow recursion. Therefore, the stack calculation is a conservative estimate, especially if there are many nodes within an SCC group and the "recursive chain" is not actually viable. However, if the design was bad and infinite recursion occurs, the stack will overflow eventually.

Other Considerations

In presenting the CFUA framework up to this point, there has been a silent assumption that all call paths have been discovered. In the C language, the use of function pointers can pose a challenge to find all of the call paths. In our customized solution, while CodeSonar does not track function pointers, it does allow direct call modeling. The downside to this approach, depending upon the prevalence of function pointer usage, may result in numerous annotations of the source code in order to model all such instances as direct calls. This problem may be compounded further if the function pointer is dynamic. In such cases, the analysis tool would need additional sophistication or the design must avoid the use of such language features. Interestingly, Jet Propulsion Laboratories Power of Ten, Rule 9 states: "*The use of pointers should be restricted,"* which speaks to this point.

Another point to consider is the uncertainty of the upper bound for worst-case stack usage. The upper bound may not be a "real" upper bound because certain paths may not be viable due to data dependencies. Considering the absence of a static analysis tool tracking such data dependencies, the upper bound, while conservative, may never be reached in practice. This is not an issue if there are sufficient resources that can be allocated for the upper bound.

Conclusions

In the example chosen for this paper, we presented a preemptive design that is representative of the homegrown task scheduler employed in our product software. It is a run-to-completion design for each preemption level, which implements a singular common stack. Therefore, the worst-case projection for our product software becomes:

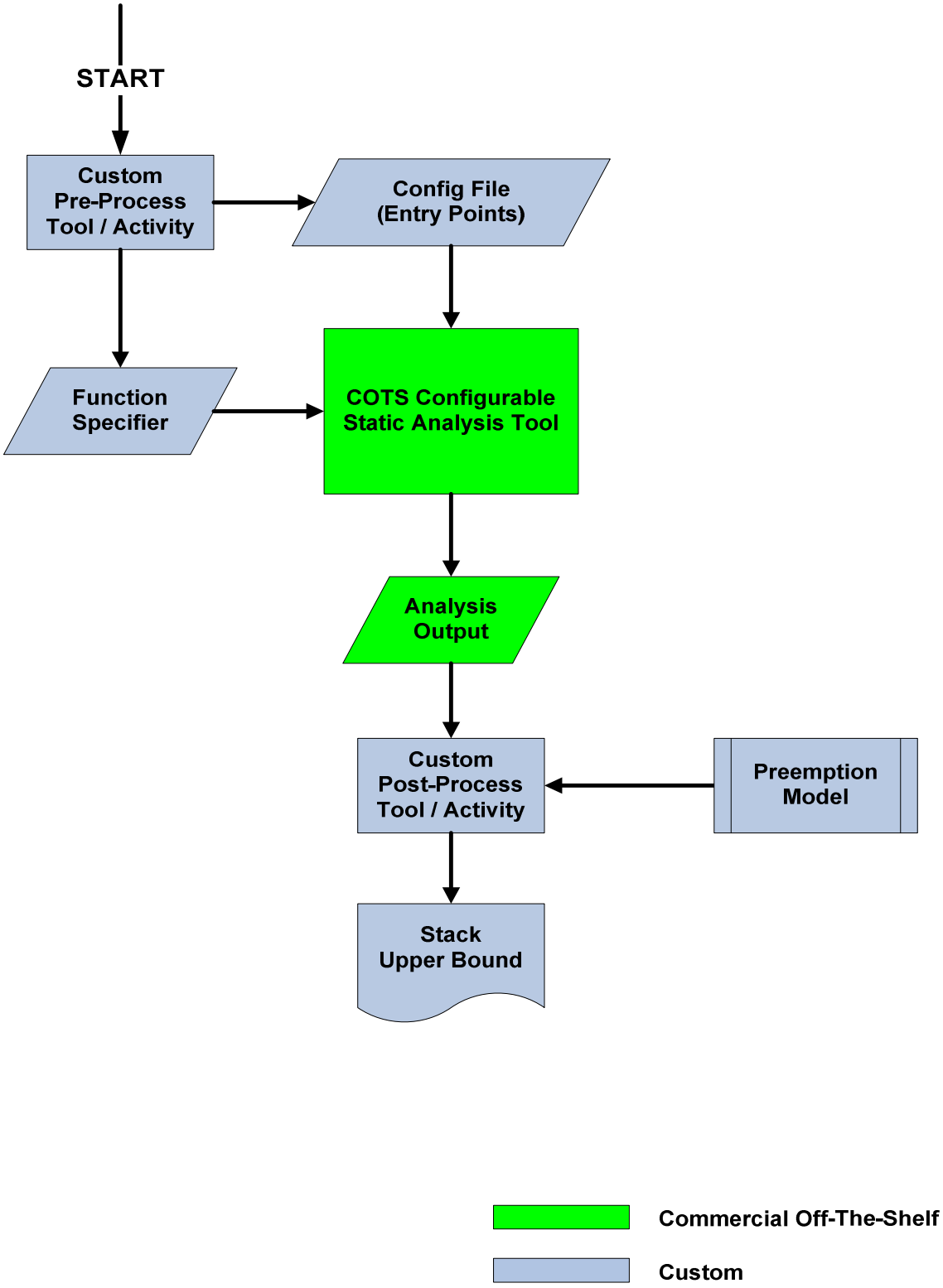
Max(Level1) + Max(Level2) +…….Max (Level N) = worst-case stack usage

In summary, establishing a lower and upper bound is a framework. Each design is different, and ultimately would have to employ different methods for calculating worst-case stack usage based on the preemption schemes in effect.

As we stated in [1], our static analyses are driven by design constraints within the framework of safety critical medical devices, which have necessitated customization. For other embedded developers, COTS tools may offer an adequate solution. With respect to stack usage analysis, while we chose the CFUA approach to establish an upper bound, using only the watermark approach may be an adequate solution for other embedded environments. If that is not the case, perhaps COTS tools such as StackX [4] can be explored before embarking on a customized approach.

Finally, some might argue that the best place to employ stack usage calculations is within the compiler itself. In the absence of that, perhaps COTS static analysis tool vendors can eventually offer an off-the-shelf solution that is “configurable.” This is a good solution since these tools typically have proven algorithms that allow them to build accurate call trees. As previously discussed, we use a customized version of CodeSonar from Grammatech, but one could envision an eventual “configurable” off-the-shelf solution from numerous tool vendors (See Figure 6). Providing an input function specifier file introduced in Figure 3 may be a good place to start. Ultimately, establishing the upper bound is probably a task best left to the post-processing of the tool’s output data, which is the model we employed. CodeSonar calculates the maximum call depth for each task thread we specify, and then we post-process the output in order to establish the true upper bound based on our preemptive design.

Figure 6: Possible Generic Framework for Future COTS Solutions



References

[1] Gerald Rigdon. Static Analysis Considerations for Medical Device Firmware. July, 2010.

[2] Jack Ganssle. The Art of Designing Embedded Systems. Elsevier, 1999.

[3] David N. Kleidermacher. Using static analysis to diagnose & prevent failures in safety-critical device designs. Published in Embedded.com. September, 2008

[4] Express Logic, Inc. StackX. http://www.rtos.com.