Optimizing Real-Time Application Software for Small Cells

Sridhar K V, Radisys

Operators are looking to deploy small cell technologies as quickly as possible to generate revenue, offload data traffic and leverage the existing macro network efficiency. These small portable platforms have limited hardware, memory and CPU speed so operators need real-time application software to manage their performance, stability, scalability and efficiency to ensure customer satisfaction.

Here’s why real-time application software is great:

There are several challenges associated with small cell technologies that present the need for real-time application software:

- Performance, such as throughput and process loading.
- Stability and maintainability of the application.
- Scalability, like operation planning and augmenting features.
- Efficiency, like dynamic event detection, analysis and avoidance.
- Efficacy, like system state predictability and outage restoration.
- Dynamic congestion management.
- Reliability.

But, it’s not a perfect solution because it’s not a perfect world:

In the development of end-to-end real-time application software, it can be a challenge to balance the need for quality while trying to meet a tight schedule. Due to the speed at which these small cell technologies are being deployed, the software is often developed aggressively without good technical planning, resulting in last minute hiccups.

BUT by following some of the basic rules of software implementation, designers can optimize their real-time software. However, in the software development business, development of end-to-end real-time application software on any platform poses a great challenge in terms of quality and timing.

Optimization and Optimization Techniques

Optimization is one of many desirable goals in software engineering, but it is often antagonistic to other important goals such as stability, maintainability and portability. At its most basic level, optimization is beneficial and should always be applied; but at its most intrusive level it can be an unending source of time consuming implementation and bug hunting. Therefore, it is absolutely necessary to be extremely cautious and wary of the cost of optimizing software code. However, if you follow certain basic, general rules of software design and implementation, you can bring the appropriate amount of optimization to real-time software.

Let’s take a look at some of the mandatory aspects of the software design and implementation techniques that can inherently improve the efficiency and performance of CPU and memory usage in small cell systems.

Generally a real-time application is composed of multiple tasks with different levels of criticality. Normally, in a real-time system losing deadlines is not desirable, but it is sometimes still inevitable depending on the requirements. So a real-time application can be organized in two levels; soft real-time and hard real-time.

A soft real-time application could lose some deadlines and the systems could still work correctly. However, hard real-time systems cannot lose any deadlines as that could cause fatal results in the system.

A hard real-time application will give precise and deterministic control of the system with very high time accuracy and very low latency. Hard real-time systems are provided through an RT-Linux kernel and modules. Thus, the development of hard real-time systems will resemble the development at the kernel level.

On the other hand, soft real-time applications do not provide much time accuracy. They depend on the OS scheduler and are likely to miss some deadlines. To combat this issue, developers can patch the Linux kernel in a way that minimizes scheduler latency, provides resource management and Quality of Service (QoS) and more.

The Design Paradigm

In practical situations designers striving for a perfect design may end up with no design at all, due to schedule and cost overruns. The truth is that a perfect design is often the enemy of a good design. Here’s my recommendation: it is always better to start with a simple design and then make reasonable and rational compromises during the design phase itself. This will save you from making unreasonable compromises in quality when you’re faced with looming product delivery dates.

In the real-time system design paradigm, the real-time software can be viewed as a collection of interconnected components. Depending on the context and granularity of scale, a component may be, for example, an entire system, an application, a process or a library module. It is very important to consider how these collections of the components are connected together, either loose coupled or tight coupled, which ultimately determines the performance and scalability of the system.

Keep in mind that when developing software architecture it is beneficial to prefer designs that will reuse already developed software modules. The future reusability of the software and modules should be a factor in choosing new architectures.

Also, avoid the “let’s start with a clean slate” approach for developing software or systems. New projects should build over the results of previous projects. This approach lowers cost by reducing complexity of the system you are developing.

Basic Rules of Software Implementation

Let’s look at the technical attributes that are most important for the design and development of an end-to-end real-time software application that is on a platform where the hardware resource is scarce, such as a small cell or femto platform.

Real-time application software must possess generic attributes of behavioral and functional predictability, which can be achieved only if there is a clear operational model of the application software. This operation model must define the following aspects:

- Shall possess an elegant processing or threading model.
- Shall have Software Application State model.
- Shall have Software Application Model.
- Shall have Execution State model.
- Shall provide the procedural attributes for the application.
- Shall provide the logical control attributes where the application software control can be ponderable at any instance of execution.
- Shall always define functional attributes especially for debugging the software.

Critical Design Factors

The key factor for real-time software application in regards to system performance is identifying the correct threading model. It is important that the real-time execution minimizes the exchange of information across the threads when the time sensitivity of the execution is critical. It is also important to group the executional flow of information so that all time sensitive computations are tightly coupled within a single thread. All auxiliary functions that are important for the system’s stability, but not specific to performance, can be designated to run as separate threads so they don’t intervene in the critical execution path. In addition, multi-threaded programs are harder to debug than their single-threaded counterparts. So the developer must decide during the design of the software the number of threads that are critical from the perspective of software architecture.

An example of a real-time processing model can be seen below:

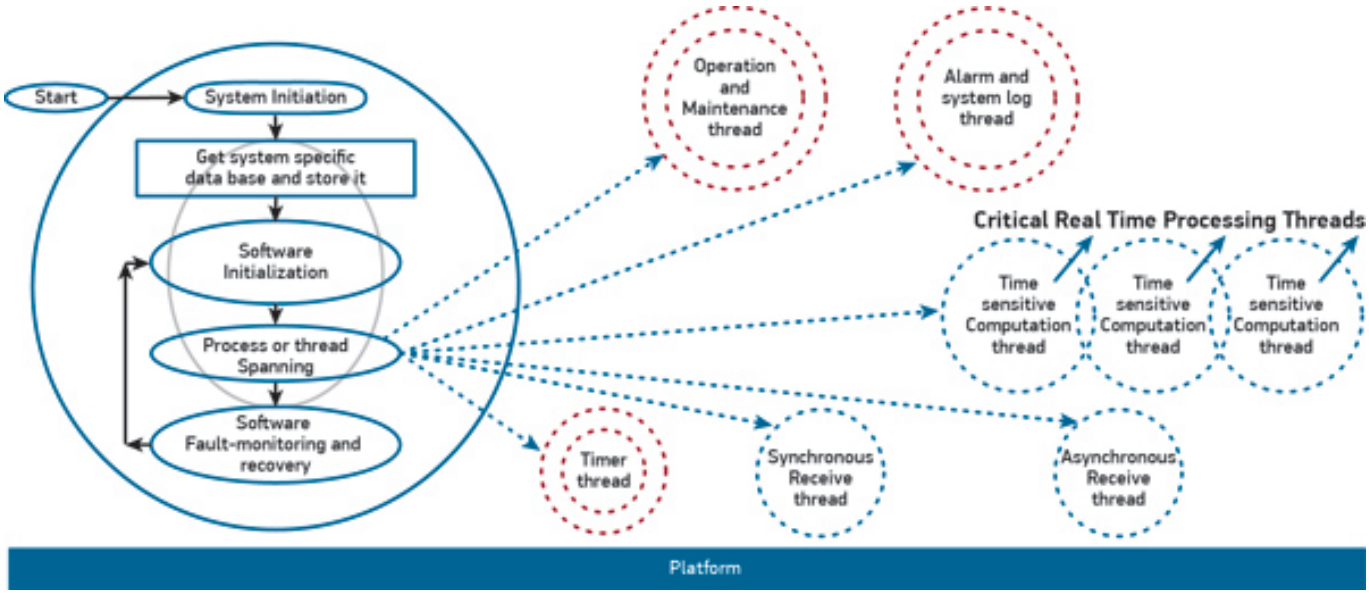

Figure 1: Real-time processing model

Real-time applications have immense behavioral diversity, therefore fixed system behavior cannot cater to many real-time application requirements, and the programmer should have ultimate control of the behavior of the system. By indicating the relative priorities for activities, a programmer can affect throughput and responsiveness goals for the system with much finer granularity. This can be achieved when the programmer selects the right scheduling policy, pre-allocation of the system, and application resources for critical services.

Another critical design factor is adopting the correct application process model. This allows the application to associate deadlines with real-time activities of the thread. In addition, the system can employ a deadline-based scheduling policy to ensure that deadlines are met or to cancel obsolete operations when deadlines are missed.

In addition, the application should be equipped with self-fault detection and recovery mechanisms. Usually small cells have a limited, well-defined set of users, so any fault can cause serious business impact in terms of customer satisfaction.

Real-Time Software Application State Model

The Software Application State Model represents the high-level grouping of software processing with respect to the external stimuli. Normally, in real-time software application, software processing states can be grouped as follows:

- System Configuration State Model (SCSM): Drives the external dependency of the software.
- Software Initiation State Model (SISM): Campaigns on the internal dependency of the software.
- Software Ready State Model (SRSM): Provides the actual entry for processing the software events or external stimuli.
- Software Execution State Model (SESM): Serves as the functional state model for all events comprising of state machines that play a major role in

defining the predictability and stability of the system.

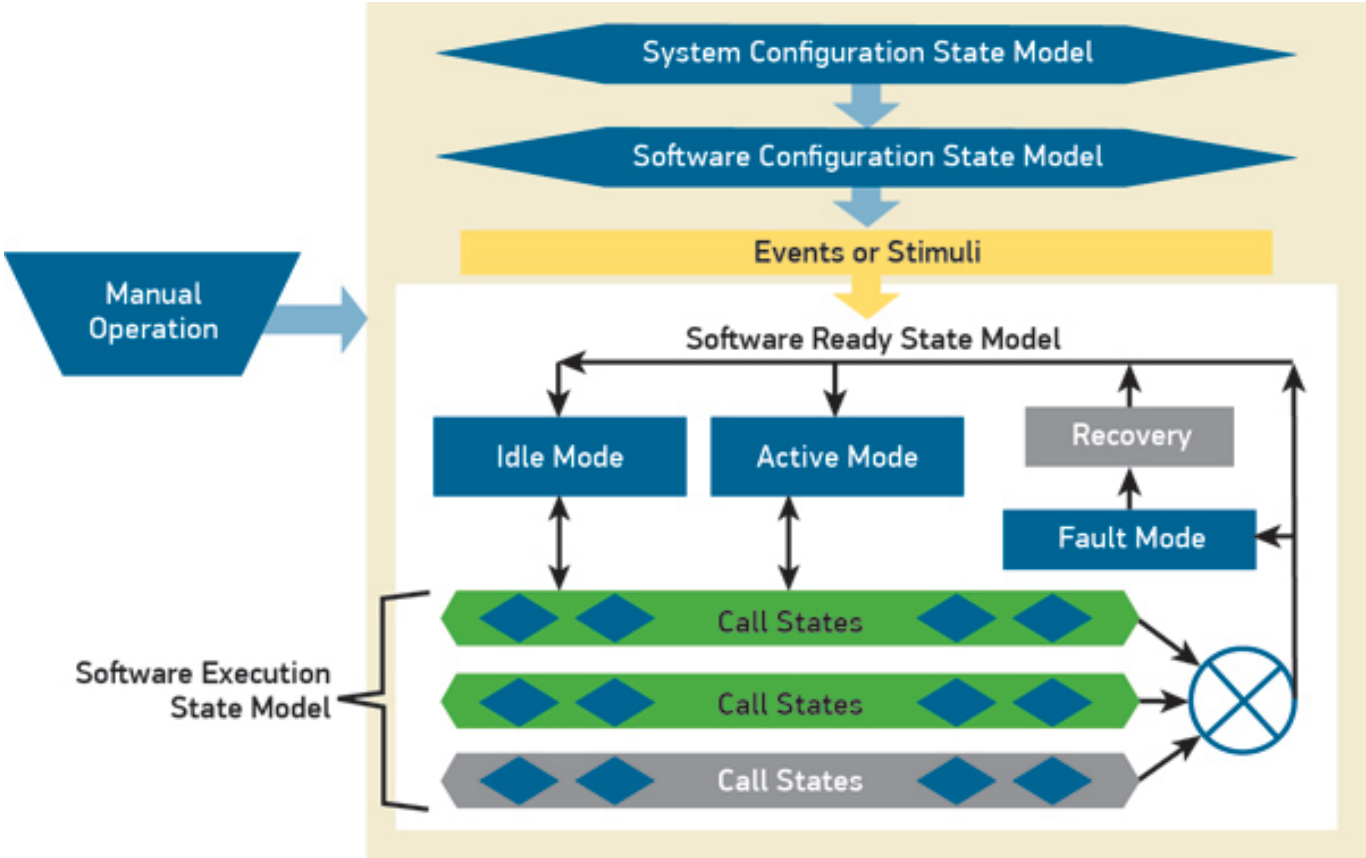

Figure 2: Real-time software application model

Why are State Machines Required?

A state machine is normally required for describing the corresponding properties and functionalities of the software; using state machines will simplify the code and also make it easier to understand the feature flow.

It is very important to manage the input and output of the state machine. If it is not managed systematically, strange things will begin to happen to the state machine in actual operation. Illegal states will be mysteriously entered, total state machine lockup or hang-up can happen and, inevitably, the software design will fail intermittently in the field.

Event data management plays a crucial role in the state machine's design. In order to manage the event data, developers need a core primordial functional entity that can evaluate the input, coordinate across various state machines, and manage the input and output of the state machine respectively. It is possible that the state machine could require that the output variables be at a unique logical pattern in two different machine states, so the coordination among the state machines necessitates a software engine. This software engine is called the Core Engine.

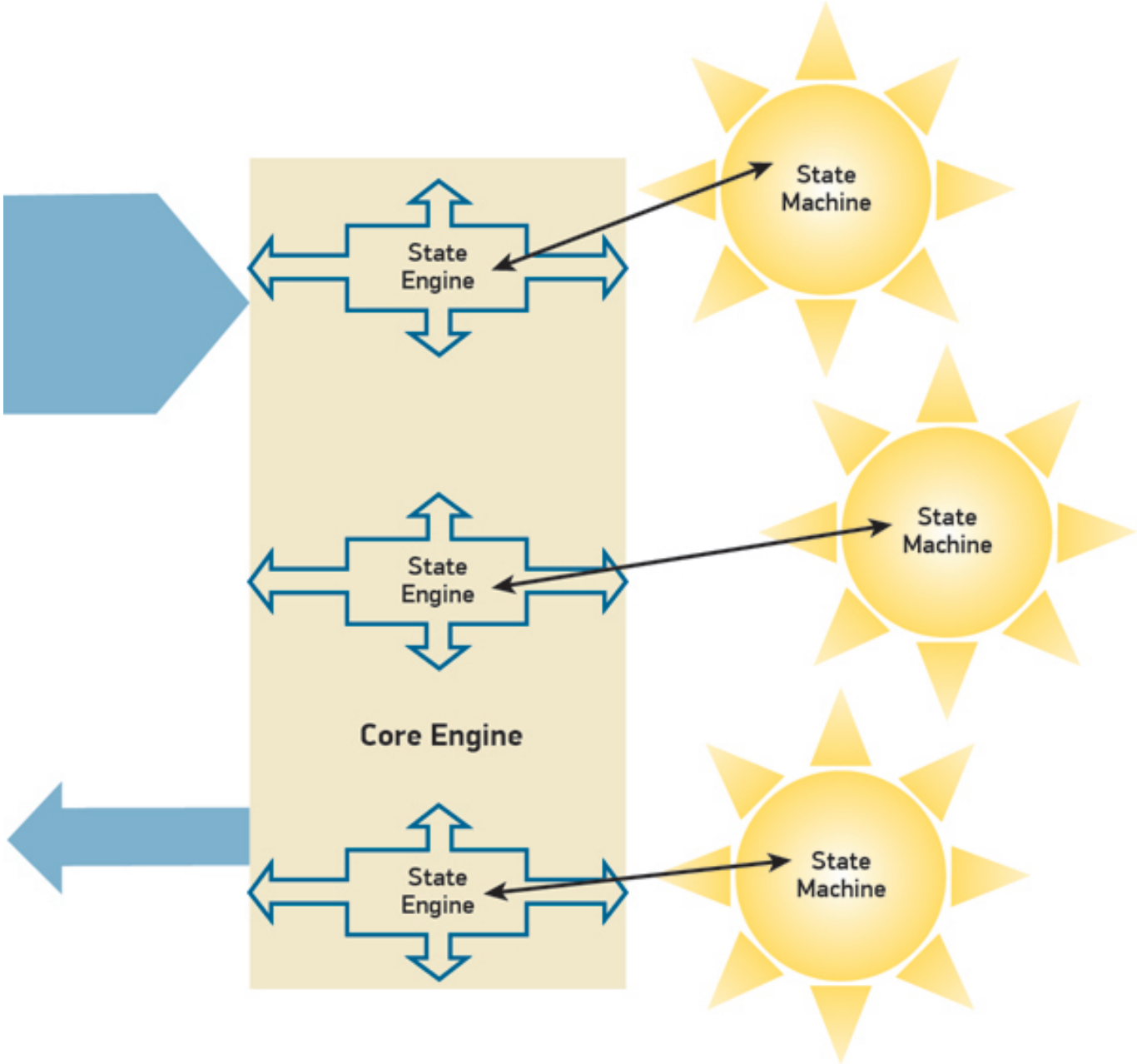

Figure 3: Overview of the State Machine

Generic Architecture of the State Machine

A generic state machine should contain the following components:

- The software event, the trigger for invoking the state machine.
- The event data, the software information for processing.
- Event data validation.
- The state transition information.
- The state.
- The handler functions.

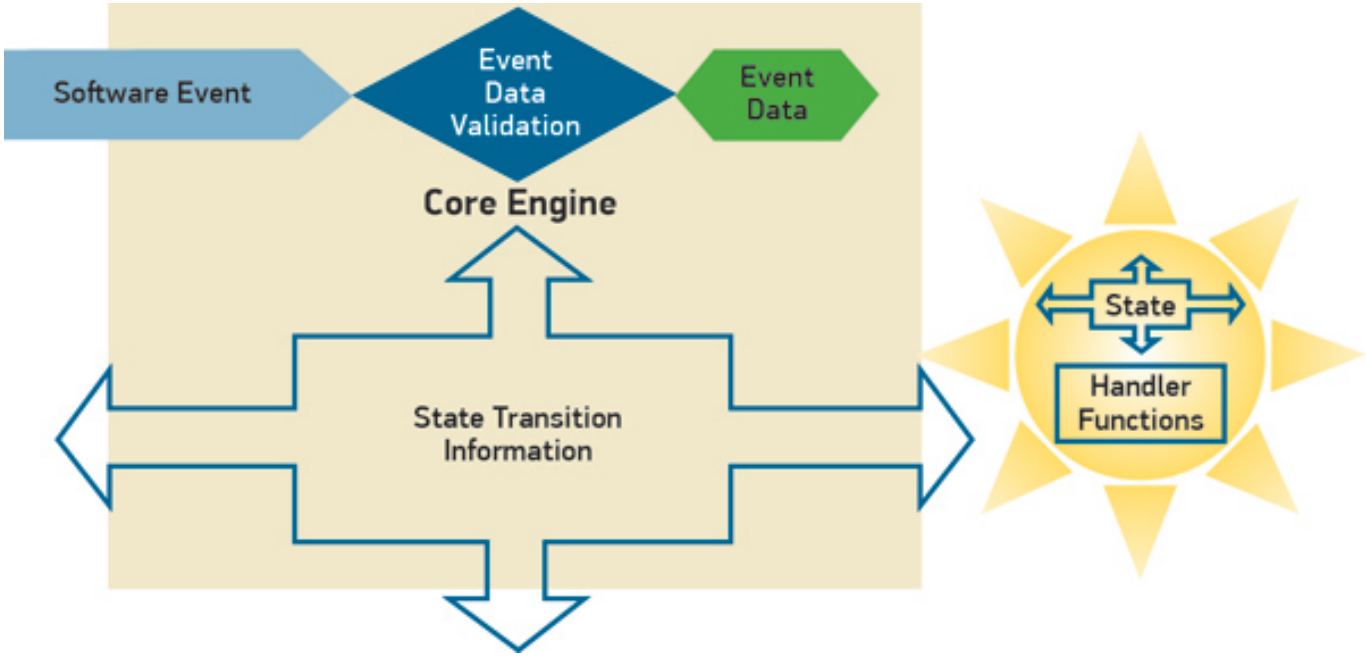

Figure 4: Generic architecture of the state machine

Let's take a closer look at these components:

The software event. This can be a simple trigger like an external signal which is responsible for a change of the system or application state.

The event data. This is the set of information associated with the event; it is associated with the current state, the configuration, or the health of the application or system.

Event data validation. This is the real functional component that can be used for validating the data before it enters the state machine. The honoring of the software event for further processing is based on the data validation.

The state transition information. This is the core part of the engine which decides whether the event can be processed in the state machine, and ensures that it generates a predictable state transition.

The state. This represents the system or application state that is processed within the state machine.

The handler functions. This component processes the event based on the event data and makes logical decisions that lead to a state transition. The state transition does not need to result in the actual change of state. The handler function may also lead to the start and stopping of timers.

State machines are a common construct in real-time systems that must complete an execution within a defined execution time budget. Each iteration of the state machine execution is associated with the execution of the current state code and the state for the next iteration. Often, this state-based operation provides an opportunity to reposition code so that it is no longer on a 'worst-case' path. This is achieved by balancing the execution time of the different states so that the worst-

case execution time is minimized.

Data Structure Framework

The design of the data structure is key for any real-time application architecture, so it is best to design with the utmost caution. Efficient data structure modeling also depends on the underlying programming language being used, and it has to be efficient and optimized.

Let’s also consider the memory aspect of the data structure modeling. When dealing with a group of unrelated data and the computational intensity is critical, then the way the elements are organized in the structure is critical from the perspective of performance.

Data Structure Alignment

Every data type in the structure will have an alignment requirement that is mandated by the processor architecture. One of the real challenges in computer architecture is not memory capacity but memory speed. If the real-time software architecture is constrained by disk and memory latency, then the system is not going to win against the performance requirements. When it is required to have access to huge amounts of data, the latency for moving it around will start to dominate software performance.

Cache Memory and Data Structure Alignment

Cache memory is an extension of the multi-level store concept. It is a small, expensive, but extremely fast memory buffer that sits between the CPU and the physical memory.

The organization of the data structure elements depends on the processing algorithms in which these structural elements are being accessed. Inefficient organization of the data structure elements can lead to intense cache processing which results in cache misses and page faults. Efficient cache-line processing of data elements that are used concurrently is one way to reduce intense cache processing. Application developers should explore different data arrangements and data segmentation policies for efficient computation.

Some tips on data structure alignment with respect to cache memory:

- When arrays are involved, the size should be power of 2.
- Group individual array elements into a structure and then index if the range is the same or close enough.
- If the data structure is big, i.e. it involves multiple embedded data structures, check the processing path and which elements it is accessing. Try grouping them into a structure or place all of them on top.
- You may need to reorganize the code instructions when data structures can’t be optimized.

- Compiler optimizations- use O3 for the final performance measurements.
- Leverage profiling data to identify cache misses.

Memory Management

Most of the time, small hardware devices, such as small cells, don’t have a storage mechanism and must therefore allocate the memory for the lifetime of the software program to hold the relevant information for execution. Once the information is stored in global memory space, the access control will become a great challenge. It is important to group the relevant information together so that clear access control is defined.

Avoiding Memory Leaks

One simple way to avoid memory leaks is to write and compile a function as a tool to track the memory allocations and print it out. The function won’t be called anywhere in the code, but it will be part of the executable, so whenever memory leaks need to be checked, one can use the functions on the gdb.

The general guideline to keep in mind is that dynamic memory allocation in real-time software should be restricted to message communication related operations. If memory is allocated statically, the design remains simple and free of memory leaks.

Debugging Abstracts

Allowing a debugger to make function calls from the debugger command line is one useful aspect while software debugging for complicated data structures. Developers should write and compile a function to traverse the data structure and print it out. The function won't be called anywhere in the code, but it will be part of the executable. It is a "debugger abstract." When we debug the code and are stopped at a breakpoint, we can easily check the integrity of the data structures by manually issuing a call to the print routine.

Coding for Debug Ability

For developers, this means breaking the system down into parts, and getting the program structure working first. Only after the basic program is working, should developers code the complicated refinements, performance tweaks, and algorithm optimizations. In the case of small devices, the platform cannot support the gdb to run for the entire software, so developers will need to compile some of the key modules with –g options so they can put gdb for some of the key functions during run time.

Profiling

Performance bottlenecks are an important factor that must be known one way or another. Normally, good compilers will have comprehensive tools for measuring performance analysis which makes this job easier.

By using profiling, developers will be able to analyze the runtime behavior of the program:

- Which parts (functions, statements, etc.) of a program take how long?
- How often are functions called?
- Which functions call which?
- Memory consumption.
- Memory accesses.
- Memory leaks.
- Cache performance.

The two kinds of profiling are Invasive Profiling and Non-Invasive Profiling.

In the case of Invasive Profiling, developers need to modify the program (code instrumentation) and insert calls to functions that record data.

Advantages of Invasive Profiling:

- It is very precise.
- Theoretically at the instruction level.
- Precise call graph.

Disadvantages of Invasive Profiling:

- Potentially very high overhead.
- Depends on the instrumentation code that is inserted.
- Cannot profile already running systems.
- Can only profile application (not complete system).
- For small devices, it may not be practicable.

In the case of Non-Invasive Profiling, we get statistic-sampling of the program. It is required to use a fixed time interval or hardware performance counters (CPU feature) to trigger sampling events and record instruction pointer at each sampling event.

Advantages of Non-Invasive Profiling:

- Small overhead.
- Hardware assisted.
- Can profile the whole system (even the kernel!).

Disadvantages of Non-Invasive Profiling:

- Not precise (“only” Statistics data)

- Call Graph possibly not complete some functions are never sampled

Conclusion

In developing real-time application software for small cell platforms, it is important to realize that performance optimization while in complete optimization in most of the cases is impossible, there is always room left to optimize. Developers must understand the nature of their specific optimization goal, and then bound it by the input/output performance and the best possible algorithm in the middle. By following some of the basic rules of software implementation outlined above, designers can optimize their real-time software.

For more information visit http://go.radisys.com [1].

Source URL (retrieved on *09/18/2014 - 6:49am*):
http://www.wirelessdesignmag.com/articles/2013/02/optimizing-real-time-application-software-small-cells?qt-most_popular=0&qt-blogs=0

Links:
[1] http://go.radisys.com