

DesignCon 2005

RTOS Synthesis to Reduce Power Consumption

Bob Zeidman, Zeidman Technologies, Inc.
(408) 255-9279
bob@zeidman.biz

Abstract

This paper introduces the concept of software synthesis, which is analogous to hardware synthesis. It explains the concepts and advantages of a tool that automatically generates complex software source code based on input from a user. This paper describes in particular how RTOS synthesis can reduce system power consumption by targeting a smaller, lower power processor than would otherwise be required. An example is given showing how an RTOS was synthesized for a multitasking Web server running on a 32-bit NIOS processor in an Altera FPGA then re-synthesized for a low-power 8-bit Cypress PSoC processor.

Author Biography

Bob Zeidman is president of Zeidman Technologies. Bob has designed ASICs, FPGAs, and PC boards and written firmware for many types of embedded systems. He is the author of the textbooks *Verilog Designer's Library*, *Introduction to Verilog*, and *Designing with FPGAs and CPLDs*. Bob holds a patent in software synthesis and has an MSEE from Stanford, and a BSEE and a BA in physics from Cornell.

SOFTWARE SYNTHESIS

Synthesis is the process of taking a high-level description and turning it into a lower-level description that, in the case of software, can be compiled directly. By working at a higher level, the user is kept uninvolved with implementation details. In particular, this paper deals with synthesis of a real-time operating system (RTOS) that is used to control multiple tasks running on an embedded system. Although RTOS synthesis has many advantages for embedded systems design, including shortening development time and producing much smaller footprint code, this paper focuses specifically on the low power optimization that can be achieved through RTOS synthesis.

THE PROBLEM

Currently, an embedded system designer has two options when it comes to an RTOS – purchase an off-the-shelf system from an RTOS vendor or write a custom one.

Writing an RTOS, even a simple one, requires operating system expertise in such things as multitasking, mutexes, and interrupt handling. The designer must be aware of conditions like race conditions, deadlock situations, and priority inversion. Writing and debugging an RTOS requires effort that could be better spent on the core technologies of the embedded system – the proprietary drivers for custom hardware and proprietary algorithms and application tasks running on top of the RTOS.

Using an off-the-shelf RTOS from an RTOS vendor has its advantages, but also its disadvantages. Advantages include the fact that the RTOS is tested and supported by the vendor. Disadvantages include the fact that the system is difficult to debug because much is hidden from the user. It also requires a large memory since it needs to support all possible users with various kinds of applications, and so it incorporates features that each particular system may not be using. Although the RTOS is optimized for the particular processor to which it is ported, that is often not the processor that is ideal for the application and is probably not the most low-power processor for the application. This is because RTOS vendors have an interest in porting their RTOS only to high-end processors so that they can charge more for these tools. RTOS vendors, other than the very smallest ones, do not port tools to simple low-power processors. Despite this, systems using 8-bit processors are far more abundant than those using 32-bit or 64-bit processors.

Another factor that discourages RTOS vendors from porting their RTOS to smaller processors is that these processors do not have the hardware necessary to assist with task management and task switching. In particular, off-the-shelf RTOSes need a memory manager to keep tasks from interfering with each other. An off-the-shelf RTOS also relies on task-switching hardware in the processor to allow the state of the machine to be swapped out and a new state swapped in during a task switch. Smaller processors do not incorporate these features.

So writing your own RTOS is complex and time consuming while purchasing an off-the-shelf RTOS can require a processor that is more complex, costly, and power hungry than is otherwise necessary.

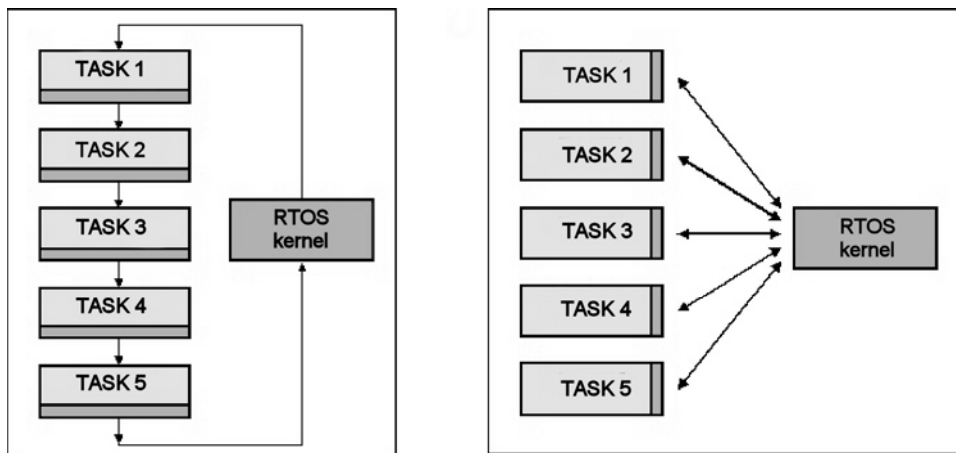
THE SOLUTION

RTOS synthesis provides a good way to create an RTOS for these smaller processors even though they lack the hardware support and have limited resources. The system designer writes drivers and application tasks in a high level language like C. Instead of complex system API calls, the programmer uses “primitives” that look like simple C function calls. The programmer then enters some system information into the RTOS synthesis tool. Detailed information about the particulars of the primitives and system information used for RTOS synthesis are given in the next section. A general diagram of software synthesis is shown in Figure 1.



Figure 1. Software synthesis

Once the system information is entered and the code is written, the synthesis tool examines all of the source code. Based on the system requirements and the particular primitives used in the application tasks, the tool strips out the primitives in the source code and replaces them with source code routines to manipulate operating system data structures, set mutexes and semaphores, open and close mailboxes and message queues, and call operating system functions.





KEY:  Task management code
 User code

Figure 2. RTOS synthesis

Next the synthesis tool writes the task management kernel of the RTOS, again in source code. The kernel includes only the functionality that is required by the set of tasks. Data structures used for global variables, semaphores, mutexes, and task swap areas are defined to have the minimum sizes necessary to support the tasks. Various scheduling

algorithms can be specified by the user, giving an added dimension of flexibility. Figure 2 shows a diagram of a simple embedded system. The synthesized code is represented by shaded areas while the user code is represented by unshaded areas.

SYNTHOS™

In creating our own RTOS synthesis tool, Zeidman Technologies was able to boil down the necessary primitives to only five. The implementation details that follow define the primitives that are inserted by the programmer, followed by an example of a project configuration file (or “SOP file”) that contains information about the system requirements and the relationships between tasks. By using only these basic primitives and a simple text file, the entire RTOS can be synthesized and all code within each task can be synthesized to allow inter-task communication and task-RTOS communication.

SynthOS Primitives

The learning curve for SynthOS is extremely short because there are only 5 primitives:

- SynthOS_call
- SynthOS_check
- SynthOS_sleep
- SynthOS_start
- SynthOS_wait

Each of these primitives gets synthesized into C code calls to an RTOS that is automatically generated in C by SynthOS.

SynthOS_call

Syntax

```
SynthOS_call(taskname(a, b, c, ...));  
retval = SynthOS_call(taskname(a, b, c, ...));
```

Description

This statement is used to begin execution of another task (taskname in the example) while suspending execution of the current task until the called task has completed. In the first example, the called task does not return a value. In the second example, the called task returns a value.

Parameters

Taskname	This is the name of the task being called.
a, b, c, ...	These are the parameters for the called task.
retval	This is the return value for the called task.

Function

Execution of the called task is begun and execution of the current task is suspended until the called task has completed. To begin execution of a task without suspending the current task, use the SynthOS_start statement.

SynthOS_check

Syntax

```
flag = SynthOS_check(taskname);
```

Description

This statement checks whether a task is currently executing.

Parameters

taskname	This is the name of the task being checked.
flag	A Boolean variable representing whether the task taskname is currently executing (true) or not (false).

Function

SynthOS_check returns true if the task is executing, false if the task is suspended.

SynthOS_sleep

Syntax

```
SynthOS_sleep();
```

Description

This statement is used to give control back to the RTOS and resume execution at a later time as determined by the RTOS.

Parameters

None.

Function

Execution of the current task is paused. At some later point, the operating system will restart execution of that task at the point after the SynthOS_sleep primitive. When this resumption occurs is determined by the operating system according to its scheduling algorithm and the priority of other tasks waiting to execute.

SynthOS_start

Syntax

```
SynthOS_start(taskname(a, b, c, ...));
```

Description

This statement is used to begin execution of another task (taskname in the example) without suspending execution of the current task.

Parameters

taskname	This is the name of the task being called.
a, b, c, ...	These are the parameters for the called task.

Function

Execution of the called task is begun and execution of the current task continues immediately after the primitive is executed. To begin execution of a task while suspending the current task, use the SynthOS_call statement.

SynthOS_wait

Syntax

```
SynthOS_wait(taskname);
```

```
SynthOS_wait();
```

Description

This primitive suspends execution of the current task and waits for another task to finish executing or for a condition to be true. The condition can be any legal C condition such as $(x == i*j + 5)$.

Parameters

taskname	This is the name of the task being checked.
	This is any legal C expression such as $(var1 * \sin(x) < 5)$.

Function

Execution of the current task is suspended until the specific task is idle or the condition is true. Note that this statement does not begin execution of taskname. To begin execution of taskname and then wait for it to finish executing, use the SynthOS_call statement.

SynthOS SOP File

The SynthOS Project file contains configuration information about the system requirements and the relationships between tasks. An example is shown in Figure 3. General information about the hardware platform and the specific tasks are entered into the SOP file to allow the RTOS synthesis tool to figure out the relationship between tasks and what kind of RTOS to implement.

```

# SynthOS Project File
[tool]
version = 1.00

# This is the start of the project section.
[project]
project_name = Project X
target = 68HC05
language = C
scheduler = round_robin
contact = Vladimir Nabokov
company = PaleFire Corporation
website = www.palefire.com
email = vlad@palefire.com
description = Mobile Phone Prototype
compiler_directives = $explicit $base10 $xyz

# The source code files are listed here
[source]
file = ConfirmDialog.c
file = ../BigTask.c
file = F:/SynthOS/Code Development/SmallTask.c

# The library object files are listed here
[lib]
file = iolib.o
file = ../lib/mathlib.o
file = C:/Zeidman/libraries/TCPIP.o

[interrupt_global]
enable = ON
maxEnable = 3
getMask = int intGetIntMask(void);
setMask = int setIntMask(void);
enableInt = void intEnable(int Vector);
disableInt = void intDisable(int Vector);
enableAll = void intEnableAll(void);
disableAll = void intDisableAll(void);
setMaxEnable = void setMaxInterrupt(int maxInt);
checkInt = bool isEnabled(int Vector);

# This defines an individual interrupt vector
[interrupt]
interrupt = 1
vector = clockTimer

# Each task has an associated [task] section.
#Task1
[task]
entry = Task1_routine
period = 1
priority = 0
TCBQ_depth = 10
type = include

```

Figure 3. Example SynthOS SOP file

LOWERING POWER

A synthesized RTOS creates OS data structures for each task in software, rather than relying on built-in hardware structures for accomplishing such things as task switching and memory management. Many high-end processors have such built-in hardware to assist the operating system and can be much faster than software for complex systems operating at worst case. For many embedded systems operating in normal conditions, the advantages of this hardware can be minimal or disappear altogether, as described below.

Yet the hardware has a cost in terms of silicon area (and thus dollar cost) and power consumption, both of which are reduced using RTOS synthesis.

Complex Processors and Hardware-Assisted Task Switching

Complex processors include large stacks and register sets. An off-the-shelf or homegrown RTOS is unaware of how many tasks may be running or how much memory, stack space, or the number of registers each task requires. Therefore, the RTOS must save the entire stack and all registers to memory during a task switch. Then the RTOS must restore the stack and registers required by the next task to run. This is a time-consuming process. To speed it up, most complex processors have specialized hardware to accomplish the task swap. This requires the RTOS to be aware of the hardware in order to use it, complicating the process of porting it to different processors. With regard to power consumption, extra hardware means extra power consumption.

Typical RTOSes also require memory management hardware that is used to protect one task from overwriting data used by another task. Again, this means more complex hardware, which translates to difficulty porting the RTOS and still more power consumption.

Simple Processors and Software Task Switching

An RTOS synthesis tool is aware before compile time exactly which tasks will be running on the system and how much memory and how many registers each task will use. The RTOS synthesis tool creates data structures in software for storing task states. Software saving and restoring of task states is generally slower than hardware-assisted task swapping, but most tasks in a typical embedded system use a small number of registers and stack space. Each software task swap space is optimized for the individual task and usually requires only a few registers and memory locations to be swapped out compared to a hardware-assisted task swap that stores every possible register because the hardware and the RTOS do not know how much is actually being used by any given task. In most cases, the optimized software task switching provided by RTOS synthesis will be faster than hardware-assisted task switching. RTOS synthesis allows the use of a processor that does not have hardware for task swapping, thus conserving power.

The same is true for memory management hardware. The RTOS synthesis tool can create separate locations in memory for each task and ensure that there is no overlap by writing the C to code to ensure this. Thus for most embedded systems, processors without memory management hardware can be used, again reducing power consumption.

Finally, since RTOS synthesis allows 8-bit processors to support an RTOS that could not otherwise do so, designers will find that many applications can be run on these smaller processors rather than larger 16-bit, 32-bit, and 64-bit processors. Task latency is still low for these smaller processors because of the optimizations that the RTOS synthesis tool performs. These smaller processors are much less power hungry, resulting in further savings in power consumption.

WEB SERVER

To demonstrate the effectiveness of RTOS synthesis, C source code for a multitasking Web server was developed for a 32-bit NIOS processor embedded in an Altera Cyclone FPGA. Using SynthOS, this code was synthesized for the processor and successfully compiled and executed. The original pre-synthesis code was then synthesized and compiled for the low power, 8-bit Cypress PSoC. The RTOS object code fit into only 1.2K bytes of ROM space while the RTOS data structures needed only 180 bytes of RAM.

CONCLUSIONS

Software synthesis allows programs to be written at a much higher level while hiding the implementation details from the programmer. Software synthesis can perform many kinds of analysis and insert many kinds of data structures to allow complex testing and debugging before compilation rather than relying on testing at run-time as is required for traditional bring-up of embedded systems. Many embedded systems are being forced onto overly complex, power-hungry processors. With the advent of RTOS synthesis, and particularly SynthOS from Zeidman Technologies, many embedded systems can be designed for simple processors, resulting in faster development times, better code optimization, easier ability to debug, lower hardware costs, and much less power consumption.

REFERENCES

1. Ball, Richard, "Cost Sensitive MCUs Designed for Niche Markets," *Electronics Weekly*, December 5, 2003.
2. Barr, Michael, *Programming Embedded Systems in C and C++*. Sebastopol, CA: O'Reilly and Associates, 1999.
3. Cataldo, Anthony, "8-bit MCUs chase high-end features," *Embedded.com*, Dec 2 2003.
4. Ganssle, Jack and Barr, Michael, *Embedded Systems Dictionary*. San Francisco, CA: CMP Books, 2003.
5. Zeidman, Bob, "The Future of Programmable Logic," *Embedded Systems Programming*, October 2, 2003.
6. Zeidman, Bob, *Introduction to Verilog*. Piscataway, NJ: Institute of Electrical and Electronic Engineers, 2000.
7. Zeidman, Bob, *Verilog Designer's Library*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1999.
8. *Nios Development Board Reference Manual, Cyclone Edition*. San Jose, CA: Altera Corp., 2003.
9. *Nios Development Kit, Cyclone Edition: Getting Started User Guide*. San Jose, CA: Altera Corp., 2003.
10. *SynthOS Users Guide*, Cupertino, CA: Zeidman Technologies, Inc., 2003.