

# SynthOS™

## Users Guide



# Table of Contents

<b>INTRODUCTION .....</b>	<b>1</b>
About SynthOS .....	1
The SynthOS Concept.....	1
<b>SYSTEM REQUIREMENTS .....</b>	<b>3</b>
<b>INSTALLATION AND SETUP.....</b>	<b>4</b>
Installing SynthOS .....	4
Running the License Manager .....	4
<b>TASK TYPES .....</b>	<b>5</b>
Task Types .....	5
<b>CODE SYNTAX.....</b>	<b>6</b>
SynthOS Primitives.....	6
SynthOS_call.....	6
SynthOS_check .....	7
SynthOS_sleep .....	7
SynthOS_start.....	8
SynthOS_wait.....	8
<b>USING SYNTHOS .....</b>	<b>10</b>
Synthesizing .....	10
RTOS Scheduling Algorithm .....	10
SynthOS Project File.....	10
Makefile Format .....	14
<b>CODING EXAMPLES.....</b>	<b>15</b>
A Simple Loop Task—“Hello, world”.....	15
Two Tasks—“Hello” and “World” .....	16
Call Tasks.....	17
Waiting on Conditions .....	18
Synchronizing with Interrupt Service Routines.....	19
Launching Interrupt Service Routines (ISR Tasks) .....	20
Timing Control .....	21
<b>INDEX .....</b>	<b>23</b>

# Introduction

## About SynthOS

This Users Guide will introduce you to a powerful program called SynthOS, a software synthesis tool for creating real-time operating systems (RTOSes) that have a very small footprint and are low cost, easy to maintain, easy to debug, and are optimized for your application.

For years, hardware engineers have had synthesis tools that have allowed them to create high level designs in familiar hardware description languages (HDLs) like Verilog and VHDL. These tools allow them to write code without concern for low level details. This code is then synthesized into a lower level code which creates the gate level descriptions from these high level descriptions. The output is still in the familiar HDL so that it can be examined and optimized if necessary. The same simulation and timing analysis tools that work on the high level description work on the low level, synthesized description because the input language is also the output language.

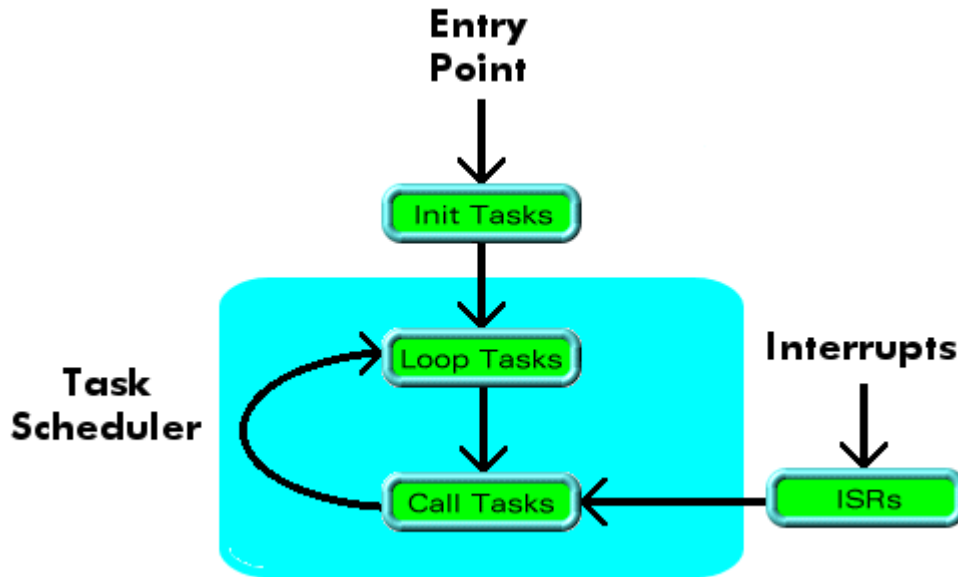
SynthOS has brought that level of design to the embedded systems programmer. By using SynthOS, you can write tasks in the C programming language to perform exactly the operations that need to be performed. You can call other tasks by using very simple SynthOS statements called "primitives." When you have written all of your tasks, you specify certain parameters for the system and for each individual task such as task priority and task frequency. Then simply run SynthOS, which will synthesize new C source code for each task, creating the appropriate flags and semaphores and modifying your original C code in order to eliminate race conditions and deadlock situations. It will also synthesize an RTOS to manage the execution of the tasks. The resulting source code files will be in C so that you can use the same compilers, debuggers, linkers, loaders, and design environment that you would normally use. SynthOS also generates a makefile for the gcc compiler that can be used to compile all the project code, including the RTOS generated by SynthOS.

## The SynthOS Concept

The concept behind SynthOS is to allow you to create very small, flexible, optimized operating systems for embedded systems, in the C programming language that is easily portable to different processors and different hardware systems. This means that you are not tied in to a proprietary RTOS; you are not forced to rely on object code of third-party source code that is difficult if not impossible to debug; you don't have to pay royalties; and you don't need to reserve memory space for code that isn't directly needed for your project. In many cases, SynthOS generated code is so small and fast and with so little overhead, that you can use a smaller, less expensive processor and less memory in your system.

When you write code to be synthesized with SynthOS, you define a project that consists of all of the tasks and other code that you need. Then, you simply write each task from start to finish without worrying about semaphores or mutexes or flags to coordinate execution of the various tasks. Whenever you want to call another task or wait for a task to finish executing or check on the status of another task, simply use a SynthOS primitive.

The synthesis process does two basic things. First, it substitutes C code for all of the SynthOS primitives and rearranges the C code of your task in order to optimize it. Second, it synthesizes a task scheduler, which coordinates the execution of the various tasks in the project. The diagram below illustrates how each type of task fits into the embedded system.



Execution of the program begins at the entry point. The Init Tasks are first executed to perform all of the necessary system initialization including initializing software variables and initializing hardware. Next the task scheduler code takes over, represented by the large shaded area. Loop Tasks are tasks that are executed repeatedly by the operating system. They typically interact with the hardware and other tasks. Loop Tasks can suspend their own execution until the occurrence of some event. Loop Tasks never die; when its execution ends, the operating system will bring it back to life, restarting its execution from the beginning. Call Tasks get executed, but only if they have been called by another task. Otherwise they remain idle and do not execute. As with other systems, interrupt service routines (ISR Tasks) are executed when an interrupt occurs. Once execution of an ISR Task completes, execution of the task running at the time of the interrupt begins where it left off.

# System Requirements

SynthOS is written in Java to run on any system that supports Java. These are guidelines for minimum system requirements just for the SynthOS are:

- **Processor:** Any processor running at 200 MHz
- **Operating system:** Windows, Linux, Unix, or any other OS that supports Java
- **Memory:** 128 MB
- **Disk:** 10 MB

# Installation and Setup

## Installing SynthOS

The installer has to run from an admin account. Simply launch the automatic installer, setup.exe.

## Running the License Manager

### Obtaining a license file

1. A file containing your computer lockcode (lockcode.txt) is found in the SynthOS folder (typically C:\Program Files\Zeidman Technologies\SynthOS). Email this file to synthos@zeidman.biz.
2. A license file called lservrc will be emailed back to you.
3. Place this file in C:\Program Files\Zeidman Technologies\SynthOS\bin. Replace the old lservrc license file if it already exists.

At this point SynthOS is installed and ready for use.

### Generating and running the examples.

1. Open up a command prompt and change to directory C:\Program Files\Zeidman Technologies\SynthOS\examples\example\_name.
2. Type make\_synthos. This will execute a simple batch file that produces synthesized code in a folder named SynthOS just below the current folder.
3. Use any C compiler to compile your synthesized code.
4. Change directory to SynthOS (cd SynthOS) and launch example\_name.exe.

### Running SynthOS

1. Type SynthOS to see a list of the command line options.
2. SynthOS should be executed from the directory where the project options file (.sop) and all project C files are located.
3. SynthOS creates a new directory named SynthOS under the directory where it was launched.
4. The files in the newly created SynthOS directory should be compiled for the specific target.

# Task Types

## Task Types

SynthOS recognizes several different types of tasks that are defined below. The specific parameters associated with each task are defined in the task section SynthOS Project (SOP) file.

Note that recursive task calls are not allowed—a task may not call itself—and C functions that are not tasks may not call SynthOS blocking primitives.

### Call Task

A Call task is one that is not executed unless it is specifically started by an executing task. A `SynthOS_call()` or `SynthOS_start()` primitive is used to start execution of the Call task, though the primitive may be implicit rather than explicitly called.

### Init Task

An Init task is executed once during the initialization of the software. It can also be started by an executing task just like a Call task in which case a `SynthOS_call()` or `SynthOS_start()` primitive is used to start execution of the Init task, though the primitive may be implicit rather than explicitly called.

Init tasks cannot contain SynthOS primitives.

### ISR Task

An interrupt service routine (ISR) task is one that processes an interrupt. It only executes when a specific interrupt is received by the processor. It is important that ISR tasks be as short as possible. This is because it is typically necessary to turn off interrupts while processing an ISR task so that it is not interrupted by another ISR task. You don't want to turn off interrupts for very long, because you may miss or delay the processing of an important interrupt. An ISR task should simply set global variables and start a Call task that performs the ISR function by using an explicit or implicit `SynthOS_start()`. A `SynthOS_call()` must not be used within an ISR task or the ISR task will not complete and the system will hang.

Unlike the other tasks, SynthOS has no particular knowledge of ISR tasks, and so SynthOS primitives cannot be used to start execution, wait for completion, or check the status of an ISR task. ISR tasks must be written in C if they interact with other tasks through global variables; this typically involves writing a small amount of assembler that sets up a stack frame, calls the C ISR tasks, then returns from the interrupt. An example of this is provided near the end of this document in the Examples section.

### Loop Task

A Loop task is executed by the task management code using an algorithm defined by the scheduler that you have selected in the SOP file.

The Round Robin scheduler executes each Loop task in succession, letting each task run until it either becomes blocked waiting for an event or until it voluntarily relinquishes control of the processor by calling `SynthOS_sleep()`. Each task may be configured to execute on every pass of the scheduling loop, or it may be executed on every  $n$  number of passes. This frequency is set in the SOP file.

The Priority scheduler allows you to assign priorities to each Loop task; when the scheduler is ready to schedule another task, it will always select the task with highest priority that is ready to run (i.e., not blocked waiting for an event). If there are several task of highest priority ready to run, the scheduler will select the task that has been waiting the longest. Task priorities are set in the SOP file.

# Code Syntax

## SynthOS Primitives

SynthOS primitives are placed in the code to perform task management. For example, a SynthOS primitive is used in one task to start execution of another task. SynthOS primitives look like C language function calls and they are used in the code similar to OS system calls for a traditional RTOS. The next sections explain the syntax for each of the SynthOS primitives below.

- SynthOS\_call
- SynthOS\_check
- SynthOS\_sleep
- SynthOS\_start
- SynthOS\_wait

Note that SynthOS primitives can be inferred by SynthOS during synthesis. In those cases, the primitives are not needed but may be used for clarity within the code.

## SynthOS\_call

### Syntax:

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

### Alternate syntax:

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

### Type: Blocking

This primitive is a blocking primitive, which means that execution of the current task is suspended until some later time when a condition is met. In this case, the condition is the completion of the called task.

### 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.

In the alternative example, a SynthOS\_call is inferred by SynthOS. The user has specified that task *taskname* is a Call Task in the SynthOS project file (SOP file). Because the task returns a value, the current task cannot continue until the called task, *taskname*, has completed and returned a value.

Note that if several tasks call the same Call task, the call requests are serialized in the order in which they occur; only one instance of a given Call task may execute at a time.

### Parameters:

<i>taskname</i>	This is the name of the task being called.
<i>a, b, c, ...</i>	These are the parameters for the called task.
<i>retval</i>	This is the return value for the called task.

### Result:

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.

### Restrictions:

SynthOS\_call can only be used to start a task that has been designated as a Call Task or an Init Task.

SynthOS\_call can only be placed in the highest level of a task. It cannot be placed in the source code of a function or subroutine that is called from a task.

The parameters passed to a task cannot be structures.

## SynthOS\_check

### Syntax:

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

### Description:

This statement checks whether a Call task is currently executing.

### Type: Non-blocking

This primitive is a non-blocking primitive, which means that execution of the current task continues after the SynthOS primitive is executed.

If several different tasks call the same Call task, this primitive will return true if *any* instance of that Call task is scheduled for execution. If you would like to know if a specific instance of a Call task is executing, it is recommended that you invoke that Call task only from one other task.

### 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).

### Result:

SynthOS\_check returns true if the task is executing, false if all instances of the task have completed execution.

### Restrictions:

SynthOS\_check can only check a task that has been designated as a Call Task

SynthOS\_check can be placed anywhere in the source code.

## SynthOS\_sleep

### Syntax:

```
SynthOS_sleep();
```

### Type: Blocking

This primitive is a blocking primitive, which means that execution of the current task is suspended until the scheduler reschedules it.

### Description:

This statement is used to give control back to the RTOS and resume execution at a later time as determined by the RTOS. It may be thought of as a voluntary "preemption point" that allows the scheduler to reschedule a higher priority task that is ready to run. Inserting SynthOS\_sleep() calls in long sections of code can increase the responsiveness of your system.

### Parameters:

None.

### Result:

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.

### Restrictions:

SynthOS\_sleep can only be placed in the highest level of a task. It cannot be placed in the source code of

a function or subroutine that is called from a task.

## SynthOS\_start

### Syntax:

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

### Alternate syntax:

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

### Type: Non-blocking

This primitive is a non-blocking primitive, which means that execution of the current task continues after the SynthOS primitive is executed.

### Description:

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

In the alternative example, a SynthOS\_start is inferred by SynthOS. The user has specified that task *taskname* is a Call Task in the SynthOS project file (SOP file). Because the task does not return a value, the current task continues without suspension.

### Parameters:

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

### Result:

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.

### Restrictions:

SynthOS\_start can only start a task that has been designated as a Call Task or Init Task

SynthOS\_start can be placed anywhere in the source code.

The parameters passed to a task cannot be structures.

## SynthOS\_wait

### Syntax:

```
SynthOS_wait(taskname);  
SynthOS_wait(condition);
```

### Type: Blocking

This primitive is a blocking primitive, which means that execution of the current task is suspended until some later time when a condition is met. In this case, the task continues execution when *taskname* is idle or *condition* is true, depending on which version of the primitive is used.

### 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 expression using constants and global variables such as ( $x == i*j + 5$ ).

### Parameters:

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

### Result:

*SynthOS\_wait(taskname)*: Execution of the calling task is suspended until the specific Call task has completed. 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. If several different tasks call the same Call task, this primitive will cause the caller to block until there are no longer any instances of that Call task scheduled for execution. If you would like to block until a specific instance of a Call task completes, it is recommended that you invoke that Call task from only one place in your application.

*SynthOS\_wait(condition)*: Execution of the calling task is suspended until the specified condition is true. Note that the condition may not be true immediately after the primitive is executed. This is because multiple tasks are effectively running simultaneously. One event in one task may cause the condition to become true, which will cause the calling task to continue executing. Before the calling task can execute the next line of its code, another task may cause the condition to become false.

**Restrictions:**

`SynthOS_wait` can only wait on a task that has been designated as a Call Task

`SynthOS_wait` can only be placed in the highest level of a task. It cannot be placed in the source code of a function or subroutine that is called from a task.

# Using SynthOS

## Synthesizing

The following command can be issued from the command line to synthesize your project.

```
java SynthOS [options] projectDescriptionFile
```

### Example:

```
java SynthOS project.sop
```

### Options:

<code>-D <i>option</i></code>	Pass <i>option</i> as a define to the compiler in the makefile for the generated code. Note that all defines to be passed to the compiler also need to be passed to SynthOS so that SynthOS and the compiler are synchronized. The makefile generated by SynthOS will send these defines to the compiler.
<code>-h[elp]</code>	Print an explanation of SynthOS user command line options and exit.
<code>-I <i>path</i></code>	Add this <i>path</i> to the default search path for include files. If not specified, the default search path is the current directory.
<code>-log <i>file</i></code>	Send stdout and stderr output to <i>file</i> .
<code>-o <i>subdir</i></code>	Place the synthesized source code files in subdirectory <i>subdir</i> . If not specified, the default subdirectory is SynthOS.

### Output:

SynthOS creates a synthesized version of each .c file along with several files implementing the synthesized operating system. In addition, a makefile is generated that can be used by the gnu compiler, gcc, to compile the synthesized code. See the section of this guide about the makefile for more details.

## RTOS Scheduling Algorithm

SynthOS can be configured to implement different task scheduling algorithms, depending on the requirements of your system. The scheduling algorithm is specified in the SynthOS Project file and the various scheduling algorithms are described below.

### Priority Scheduling

The Priority Scheduler allows you to assign priorities to each Loop task; when the scheduler is ready to schedule another task, it will always select the task with highest priority that is ready to run (i.e., not blocked waiting for an event). If there are several task of highest priority ready to run, the scheduler will select the task that has been waiting the longest. The priority for each task is set in the corresponding task section of the SOP file.

### Round Robin Scheduling

The Round Robin scheduler executes each Loop task in succession, letting each task run until it either becomes blocked waiting for an event or until it voluntarily relinquishes control of the processor by calling SynthOS\_sleep(). Each task may be configured to execute on every pass of the scheduling loop, or it may be executed on every *n* number of passes. This frequency is set in the SOP file for each task is set in the corresponding task section of the SOP file.

## SynthOS Project File

The SynthOS project file is a text file defined by the user and has the name *ProjectName.sop* where *ProjectName* is the name of the project. Each field within the file assumes a default value if you do not specify an explicit value, so you need only specify those fields which are relevant to your application. Each project has its own unique SynthOS project file, an example of which, illustrating all possible fields, is shown below.

```

# SynthOS Project File
[tool]
version = 1.00

# This is the start of the project section.
[project]
project_name = Project X
target = 68HC05
processor_size = 32
language = C
scheduler = round_robin
contact = Vladimir Nabokov
company = PaleFire Corporation
website = www.palefire.com
email = vlad@palefire.com
# Description of the project. Use a "\" to continue on the next line.
description = Mobile Phone Prototype \
This is the first version. \
I hope you'll like it.
# The compiler directives. These are used to specify DEFINES and include paths needed by
# your application. Use a "\" to continue on the next line.
compiler_directives = -D FOO \
-I /usr/local/include \
-D DUMMY=5

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

# The global interrupt routines are listed in the interrupt_global section
[interrupt_global]
enable = ON
getMask = intGetIntMask
setMask = setIntMask
enableAll = intEnableAll

# Each task has an associated [task] section.
#Task1
[task]
entry = Task1_routine
period = 1          # Used by the Round Robin Scheduler, ignored by the Priority Scheduler
priority = 0        # Used by the Priority Scheduler, ignored for the Round Robin Scheduler
TCBQ_depth = 10
type = call

# Task2
[task]
entry = Task2_routine
period = 3          # Used by the Round Robin Scheduler, ignored by the Priority Scheduler
priority = 2        # Used by the Priority Scheduler, ignored for the Round Robin Scheduler
TCBQ_depth = 3
type = loop

```

## SOP comments

All characters in a line after a pound character (“#”) are ignored as comments.

## SOP tool section

The project section is preceded by the statement [tool] as shown in the example below. An example is given below.

```

# SynthOS Version
[tool]
version = [0-9].[0-9][0-9]

```

## Version

This parameter specifies the version of SynthOS that is to be used with this project file. It is specified by the keyword `version` followed by a version number.

## SOP project section

The project section is preceded by `[project]` as shown in the example below. All of the items in this section define the user's project.

```
# 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
```

### Project name

This parameter defines the name of the user's project.

### Target

This parameter defines the processor that is to be targeted by SynthOS.

### Processor\_size

This parameter defines the bit-size (e.g. 8, 16, 32, 64) of the processor that is to be targeted by SynthOS.

### Language

This parameter defines the computer programming language of the input source code and output source code.

### Scheduler

This parameter defines the task scheduling algorithm used for this project. The two legal options are currently "round\_robin" and "priority."

### Contact

This parameter defines the person responsible for the project.

### Company

This parameter defines name of the company that is creating the project.

### Email

This parameter defines the email for the person responsible for the project.

### Description

This parameter gives a multi-line description of the project.

### Compiler directives

This parameter gives a list of any compiler specific directives that need to be used to compile the code that is output from SynthOS. For example "-D DEBUG".

## SOP source section

The source code files are listed in the source section.

```
[source]
file = ConfirmDialog.c
file = AboutDialog.c
file = ../BigTask.c
file = F:/SynthOS/Code Development/SmallTask.c
```

### file

This parameter lists a source code file used in the system. The file can be listed with an absolute path or a path that is relative to the user's current working directory.

## SOP global interrupt section

This section defines global interrupt options for the system. This section is determined by lifting the appropriate section for the target processor from the Interrupt Definition File. The user then must fill in the specifics for each setting.

```
# The global interrupt routines are listed in the interrupt_global section
[interrupt_global]
enable      = ON
getMask     = intGetIntMask
setMask     = setIntMask
enableAll   = intEnableAll
```

If your system uses interrupts (specified by "enable = ON") then you must write three routines for your processor and specify them in these fields:

- [1] enableAll—a routine that enables all interrupts. This is called by the system only once after all Init tasks have executed. This routine must have the signature "void intEnableAll(void);".
- [2] getMask—a routine that disables all interrupts and returns the previous state of the interrupt system just prior to this call. This routine must have the signature "int intGetIntMask(void);".
- [3] setMask—a routine that restores the interrupt system to the state specified in the parameter. The parameter value is normally the value returned by getIntMask. This routine must have the signature "void setIntMask(int Mask);".

## SOP task section

Each task in the project must have its own section in the project file.

```
# Task1
[task]
entry = Task1_routine
period = 1
priority = 1
TCBQ_depth = 10
type = include

# Task2
[task]
entry = Task2_routine
period = 3
priority = 2
TCBQ_depth = 3
type = loop
```

### Entry

This parameter gives the name of the C function that is the entry point for the task.

### Period

This parameter tells how many loops of the main polling loop in the task management code result in a single execution of the task. For example, a value of 3 means the task executes once every 3 loops. This field is only meaningful when the Round Robin scheduler has been specified and is ignored when the Priority scheduler has been specified.

### Priority

This parameter is a number from 0 to 31 that gives the relative priority of the task with regard to other tasks. A higher number represents a higher priority. This field is only meaningful when the Priority scheduler has been specified and is ignored when the Round Robin scheduler has been specified.

### TCBQ\_depth

This represents the maximum number of instances of the task that can be executing simultaneously. This field is only meaningful for Call tasks.

## Task type

This parameter specifies the task type.

The following types are valid.

call	Call Task
init	Init Task
loop	Loop Task

## Makefile Format

After synthesis, SynthOS creates a makefile for the gcc compiler that can be used to compile all the project code, including the RTOS generated by SynthOS. Below is an example of a makefile generated by SynthOS.

```
# SynthOS autogenerated makefile - do not modify

CC = gcc -g -c -I.
LD = gcc -g
ALL = SynthOS_Final

$(ALL) : Makefile tcb_c.o SynthOS_scheduler_c.o nios_toy_c.o SynthOS_global_c.o io.o
$(LD) -o $(ALL) tcb_c.o SynthOS_scheduler_c.o nios_toy_c.o SynthOS_global_c.o io.o

clean :
rm -f $(ALL)
rm -f SynthOS_scheduler_c.o
rm -f nios_toy_c.o
rm -f SynthOS_global_c.o

nios_toy_c.o : Makefile nios_toy_out.c
$(CC) -O2 -o nios_toy_c.o nios_toy_out.c

SynthOS_global_c.o : Makefile SynthOS_global.c
$(CC) -o SynthOS_global_c.o SynthOS_global.c

SynthOS_scheduler_c.o : Makefile SynthOS_scheduler.c
$(CC) -Wall -o SynthOS_scheduler_c.o SynthOS_scheduler.c

tcb_c.o: Makefile tcb.c tcb.h
$(CC) -Wall -o tcb_c.o tcb.c
```

# Coding Examples

## A Simple Loop Task—“Hello, world”

Here's a simple application consisting of one Loop task that repeatedly prints out “Hello, world.” This is not very useful, of course, but it does demonstrate how easy it is to specify tasks and write application code: no need to write a “main()” routine, no need to allocate and initialize kernel data structures, no need to get the scheduler running... SynthOS does all of that for you.

```
// This is stored in file "helloWorld.c"
void helloWorldTask() {
    for (;;) {
        printf("Hello, world\n");
    }
}
```

The project file (SOP file) might look like this:

```
# Project file for "helloWorld"
# This file is "helloWorld.sop"
[source]
file = helloWorld.c

[task]
entry = helloWorldTask
type = loop
```

To generate your application, go to the directory containing your source and project files (helloWorld.c and helloWorld.sop) and type

```
java SynthOS helloWorld.sop
```

at the command line. SynthOS will create a subdirectory called “SynthOS” containing several different .c and .h files along with a Makefile. If you change into that directory and type “make,” you will build your application. On Windows, the application will be called “a.exe”; on Unix and Linux it will usually be called “a”.

If you wish to debug your application in a visual debugger, you should compile with the “-g” option. For example, if you change to the “SynthOS” directory and type

```
gcc -g *.c
```

you can then do source level debugging using a visual debugger such as the GNU insight debugger by typing:

```
insight a
```

## Two Tasks—“Hello” and “World”

Here’s a slightly more complicated example with two tasks, one of which prints out “Hello” and the other prints out “world\n”:

```
// This is stored in file "hello.c"
void helloTask() {
    for (;;) {
        printf("Hello, ");
        SynthOS_sleep();
    }
}

// This is stored in file "world.c"
void worldTask() {
    for (;;) {
        printf("world\n");
        SynthOS_sleep();
    }
}
```

the SOP file might look like this:

```
[source]
file = hello.c
file = world.c

[task]
entry = helloTask
type = loop

[task]
entry = worldTask
type = loop;
```

The `SynthOS_sleep()` call in each of the tasks causes the caller to suspend execution and return to the scheduler. When the task gets rescheduled, it will resume operation where it left off, in this case inside the for loop. Since we’ve not specified priorities or a scheduler, SynthOS defaults to the `round_robin` scheduler. The two tasks thus take turns, ping-ponging back and forth, cooperating to print out “Hello, world” over and over.

## Call Tasks

A Call task can thought of as a dynamically created task that executes some functionality (often using SynthOS primitives) and then dies a quiet and mercifully painless death. Here's an example of invoking and synchronizing with Call tasks:

```
// CALL task
void printMessage(char *message) {
    printf("callTask1 says %s\n", message);
}

// CALL task
int increment(int input) {
    return input + 1;
}

// LOOP task
void loopTask() {
    int count = 0;
    for (;;) {
        // Start a Call task, but continue execution.
        SynthOS_start(printMessage(count));
        // Start another Call task, but wait until it completes.
        count = SynthOS_call(increment(count));
        printf("count = %d\n", count);
        // Wait for the increment Call task to complete.
        SynthOS_wait(increment);
    }
}
```

Can you predict the order in which the print statements will execute? And what they will print? Try it and see.

## Waiting on Conditions

The `SynthOS_wait(condition)` primitive allows tasks to synchronize with each other. Most RTOSes supply synchronization primitives such as semaphores, events, mailboxes, message queues, etc. for synchronization, but SynthOS lets you use simple C expressions to accomplish the same thing. Here's an example where the second task must not execute until the first task says it's OK:

```
int counter;

// INIT task - this will be called before any LOOP task executes.
void initialize() {
    counter = 0;
}

// LOOP task - increments counter.
void task1() {
    for (;;) {
        counter++;
        SynthOS_sleep();
    }
}

// LOOP task - waits for counter to reach a value before it executes.
void task2() {
    for (;;) {
        SynthOS_wait(counter == 5);
        printf("I'm alive!\n");
    }
}
```

Try compiling and executing this and see what happens. At first it will seem like magic—how can the second task possibly know when to wake up?—but SynthOS makes sure that happens. Internally SynthOS uses synchronization primitives to accomplish this just like other RTOSes, but spares you the pain of having to allocate, initialize and interface with them. You just program in C.

## Synchronizing with Interrupt Service Routines

Sooner or later you will want to have a Task block until some external event occurs; then when the event actually happens, you want to get the Task running to handle it. There are many ways of accomplishing this, with the method of choice depending on the primitives offered by the operating system that you're running on. Here's a textbook solution for accomplishing this using semaphores with the classic P() [wait] and V() [signal] operations:

```
// Conventional RTOS interrupt synchronization using semaphores.

// Declare a global semaphore that the Task and ISR will share.
semaphoreType *mySemaphore;

main() {
    semaphoreType *mySemaphore = allocateSemaphore();
    initSemaphore(mySemaphore, 0);
    // Other initialization code follows...
    .
    .
}

// Task that wants to be notified when an event occurs.
void task() {
    for (;;) {
        // Block on the semaphore.
        semaP(mySemaphore);
        // Handle the event.
        .
        .
    }
}

// Interrupt service routine that wants to wake up a task.
void isr() {
    semaV(mySemaphore);
}
```

In SynthOS you would do the same thing using only C expressions and SynthOS\_wait():

```
// Declare a global variable that the Task and ISR will share.
// This should be initialized in an Init task.
int myEvent;

// Task that wants to be notified when an event occurs.
void task() {
    for (;;) {
        // Wait for the event.
        SynthOS_wait(myEvent == 1);
        // Handle the event.
        :
    }
}

// Interrupt service routine that wants to wake up a task.
void isr() {
    // Toggle the event to wake up the task.
    myEvent = 1;
    myEvent = 0;
}
```

## Launching Interrupt Service Routines (ISR Tasks)

Interrupt service routines must be written in C if they write global variables that are included in a SynthOS\_wait(expression) call. Such routines must usually be launched by writing a small amount of assembler code that is dependent on the target processor and the compiler that you are using. Here's an example of the code needed to launch a C language interrupt service routine on a Hitachi H8 processor using the gcc cross compiler:

```
__asm__( "\n\  
.text\n\  
.align 1\n\  
.global _systime_handler\n\  
_systime_handler:\n\  
    push r0          ; create a stack frame for C\n\  
    push r1\n\  
    push r2\n\  
    push r3\n\  
    jsr _clockInterrupt ; call the ISR\n\  
    pop r3          ; pop the stack frame\n\  
    pop r2\n    pop r1\n    pop r0\n    rts\n\  
");
```

## Timing Control

You may have noticed that SynthOS includes no primitives for timing, such as “sleep(duration)”. Timing is often very important for embedded systems, so how can you implement them if SynthOS doesn’t offer timing services? Easy—just write your own in C. Here’s an example of a task that needs to run periodically, every 10 “ticks” of the system clock. The interrupt service routine will usually need a small amount of assembler to be launched (for example, to set up a stack frame), but the bulk of it can be written in C. Here we write our own interrupt handler for a clock interrupt that simply increments the system time. Implementing “sleep()” requires only two lines of code:

```
int systemTime;    // Global time, counts "ticks".

void clockInterruptHandler() {
    systemTime++;
}

int wakeupTime;

// Task that wants to be awakened every 10 "ticks".
void task() {
    for (;;) {
        wakeupTime = systemTime + 10;
        SynthOS_wait(systemTime == wakeupTime);
        // Do work here...
    }
}
```

Copyright Information  
SynthOS software is copyright 2001-2005 by



4950 Hamilton Ave. Suite 210  
San Jose, CA 95130  
[www.zeidman.biz](http://www.zeidman.biz)

SynthOS incorporates ANTLR parsing software developed by Professor Terence Parr at the University of San Francisco.

# Index

## A

ANTLR..... 22

## C

Call Tasks ..... 1, 5, 6, 7, 8, 13, 21  
checkInt ..... 10, 13

## D

disableAll..... 10, 13  
disableInt..... 10, 13

## E

enable..... 10, 13  
enableAll..... 10, 13  
enableInt ..... 10, 13

## G

gcc ..... 1, 3, 10, 14  
getMask ..... 10, 13

## H

HDLs ..... 1

## I

Init Tasks ..... 1, 5, 6, 7, 8, 13  
interrupt\_global ..... 13  
ISR Tasks..... 1, 5, 21

## J

Java..... 3  
JDK..... 3

## L

Loop Tasks..... 1, 5, 13

## M

makefile ..... 1, 10, 14  
maxEnable ..... 10, 13  
mutexes..... 1

## N

no\_comment..... 10  
no\_expand..... 10  
no\_global ..... 10  
no\_make..... 10  
no\_renumber ..... 10  
no\_wait ..... 10

## P

primitives .....	1, 5, 6, 7, 8, 21
Priority .....	5, 10, 11, 13
project_name .....	10, 12

## R

Round Robin .....	5, 10, 11, 12, 13
RTOS .....	1, 6, 7, 10, 14

## S

scheduler .....	1, 2, 5, 7, 10, 11, 12, 13, 14, 15, 16
setMask .....	10, 13
setMaxEnable .....	10, 13
SOP .....	5, 6, 8, 10, 11, 12, 13
SynthOS_call .....	6
SynthOS_check .....	7
SynthOS_global.c .....	10
SynthOS_global.h .....	10
SynthOS_scheduler.c .....	10
SynthOS_scheduler.h .....	10
SynthOS_sleep .....	7
SynthOS_start .....	8
SynthOS_wait .....	8

## T

Task Context Blocks .....	10, 13
tcb.c .....	10
tcb.h .....	10
TCBQ_depth .....	10, 13

## V

vector .....	10
Verilog .....	1
VHDL .....	1

## Z

Zeidman Technologies .....	22
----------------------------	----