SynthOS®

User's Guide

ZEIDMAN TECHNOLOGIES

SynthOS

ZEIDMAN TECHNOLOGIES

Contents

SynthOS ZEIDMAN TECHNOLOGIES

Introduction

About SynthOS

This User's 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 foreach 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.

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 into 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 do not have to pay royalties; and you do not need to reserve memory space for code that is not 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. Figure 1 illustrates how each type of task fits into the embedded system.

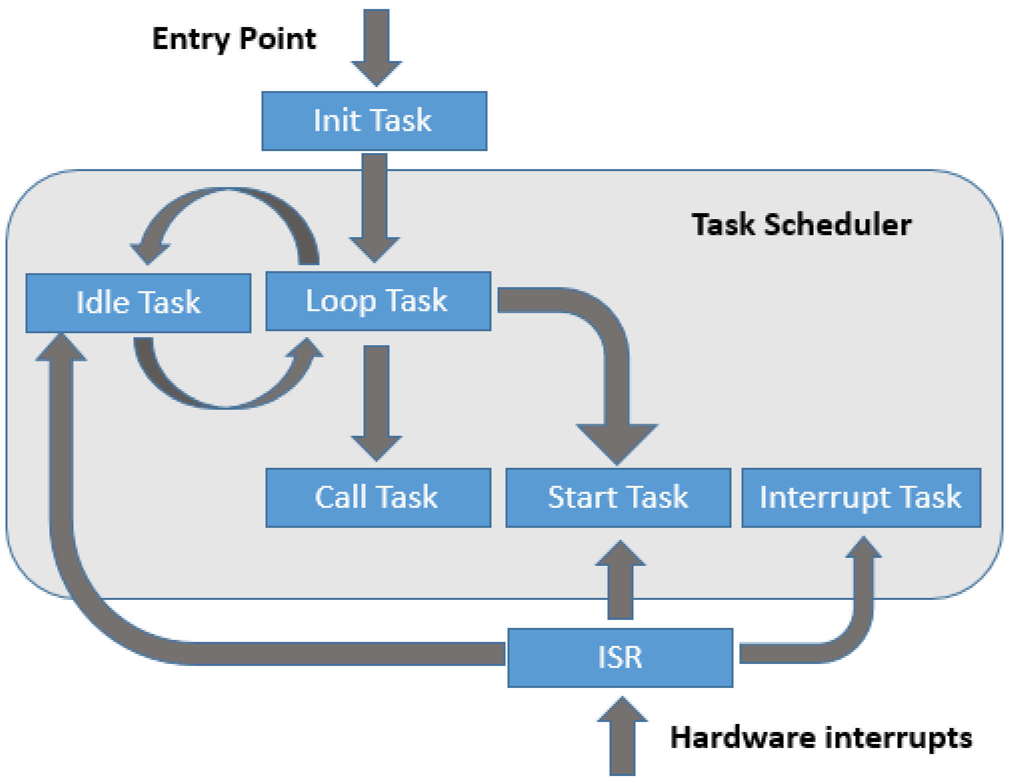


Figure 1. SynthOS block diagram

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 (ISRs) are executed when an interrupt occurs. Once execution of an ISR 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:

- **Operating system**: Windows, Linux, Unix, or any other OS that supports Java
- **Memory**: 128 MB
- **Disk**: 10 MB

Installation and Setup

Installing SynthOS

Follow the installation instruction in the SynthOS Installation Guide.

Generating and running the examples.

Examples for SynthOS project can be found on our website at www.zeidman.biz/download.

Running SynthOS

1. Type `SynthOS -h` to see a list of the command line options.
1. SynthOS as a default creates a new directory named SynthOS under the directory where it was launched, unless changed using the "-o" directive of SynthOS.
2. The files in the newly created SynthOS directory should be compiled for the specific target. SynthOS also generates a dependency file called `dependencies.mk`. This is not a make file but could easily be included in a make file to ensure all the dependencies are correct.
3. Compile the files in the SynthOS directory with the C compiler for your target processor to generate the executable or a HEX file to be loaded to the target.

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 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 and Start Task

Call tasks and Start tasks are not executed unless they are specifically spawned by an executing task. A `SynthOS_call()` or `SynthOS_start()` primitive is used to start execution of a Call task and a Start task, respectively. If a task is implicitly called (as a function), it will be executed as a Call task.

The user does not have to explicitly declare a Call task or a Start task in the SOP file; SynthOS will analyze the code and automatically detect the task type. However, if a started task can have multiple instances running concurrently, then it must be declared in the SOP file along with a `maxInstances` parameter. If multiple instances of start tasks are to be executed serially, the `maxRequests` parameter should define the maximum number of tasks that are allowed to be called simultaneously. A combination of concurrent and serial instances can also be defined. A task cannot be both a Call task and a Start task.

SOP file

```
[task]
entry = printMessage
maxInstances = 2
maxRequests = 3
```

Code

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

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

// LOOP task
void loopTask()
{
    int count = 0;

        // Spawn a Start task, but continue execution.
        SynthOS_start(printMessage[0](count));
        SynthOS_start(printMessage[1](count));

        // Spawn a Call task, but wait until it completes.
        count = SynthOS_call(increment(count));
        printf("count = %d\n", count);

        // Wait for the printMessage Call task to complete.
        SynthOS_wait(printMessage);
}
```

Init Task

An Init task is executed once during the initialization of the software; it can be used to reset the system parameters and set the configuration.

Init tasks cannot contain SynthOS primitives.

SOP file

```
[task]
entry = initializeSystem
type = init
```

Code

```
// Initialize task
void initializeSystem()
{
    // Initialize system variables
     .
     .
     .
}
```

Loop Task

A Loop task is executed by the task management code that is generated by SynthOS to schedule the task execution, using an algorithm defined by the scheduler that you have selected in the SynthOS project file.

Loop tasks are the main tasks in the system. They are initiated at system startup and run continuously for the duration of the system operation. A Loop task can be instantiated multiple times with each instance of the task running in parallel. The SOP file must define the number of instances of the task that will be initiated at system startup. Then the `SynthOS_instance()` primitive can be called from within the task to determine which instance is executing and perform the appropriate operation.

SOP file

```
[task]
entry = motorTask
type = loop
maxInstances = 2
```

Code

```
void  motorTask(void)
{
     if(SynthOS_instance()== 0)
     {
          configuration = Left_Motor;
     }
     elseif(SynthOS_instance()== 1)
     {
          configuration = Right_Motor;
     }

     // do stuff...
}
```

Idle Task

The Idle task runs in the background of the system. If you define an Idle task, you do not have to define the `setMaskAndSleep` function in the project file. Instead, the Idle task will be executed during the time that all other tasks are suspended, and it can handle the system sleep modes as needed. You can still call `setMaskAndSleep` from the Idle task if desired. An embedded system can be built by going directly into the Idle task after system startup and not defining any Loop tasks. In this architecture, all system functionality is supported using Interrupt, Call, or Start tasks. The Idle task can include the `SynthOS_start()` primitive, and it can start execution of a task that will take over the system for the duration of this task, before the system returns to the Idle task.

SOP file

```
[task]
entry = idleTask
type = idle
```

Code

```
void idleTask()
{
     // Set system to low power ...
     // Do idle operation ...
}
```

Interrupt Task

Interrupt tasks are initiated by a `SynthOS_interrupt` primitive and are called from an ISR. It is strongly recommended that an ISR call an Interrupt task, especially where the processing might take a long time. When an Interrupt task is initiated, the system will set the interrupt flag for the specific Interrupt task; the flag will be reset by the scheduler once the task is started. An Interrupt task runs in an interrupt enabled, but with the highest task priority. SynthOS enables initiation of multiple different Interrupt tasks and will serialize these calls in the order they are called. If the system calls the same Interrupt task multiple times, it will be executed only once; when the interrupt flag is reset, the system can initiate the Interrupt task again.

SOP file

```
# software Interrupt task
[task]
entry = softwareInterrupt
type = interrupt

# ISR for a general interrupt
[interruptGlobal]
enable = on
getMask = get_mask
setMask = set_mask
enableAll=enable_int
routine = someInterrupt
```

Code

```
// Interrupt task
void softwareInterrupt (void)
{
     printf ("softwareInterrupt\n");
     // do stuff...
}

//Interrupt service routine
void someInterrupt (void)
{
     printf ("someInterrupt\n");

     // do stuff...
     SynthOS_interrupt (softwareInterrupt);
}
```

In some cases, the scheduler for an RTOS needs to run with interrupts disabled to ensure that atomic action is completed without interruption, especially when the system is switching context between tasks. SynthOS supports this by using a number of predefined functions that the developer needs to implement, for enabling, disabling, and masking interrupts. This gives the developer complete control over the interrupt status during system execution. SynthOS enables interrupts at the entry to every SynthOS task, with the exception of an Init task and Interrupt task ensuring that the system can stop the execution of the task, if needed, to respond to a change in the system conditions.

The system also enables interrupts before it returns control to a task from a SynthOS blocking primitives `SynthOS_call`, `SynthOS_wait`, and `SynthOS_sleep`. Because SynthOS cannot guarantee keeping the interrupt status across the blocking call, the default behavior is to enable interrupts, but the developer still retains full control of the interrupt status and can manage the interrupt status on the return from the blocking call or by using the interrupt enable function.

ISR Definitions

ISRs are not considered to be tasks as they were in previous versions of SynthOS. All ISR are defined in the [`interruptGlobal`] section; the ISR can simply be listed as a routine. Another change the interrupt definitions is this version is that `setMaskAndSleep` function does not have any parameters, as shown in Example 1.

SOP file

```
# ISR for a general interrupt
[interruptGlobal]
enable = on
getMask = get_mask
setMask = set_mask
enableAll=enable_int
setMaskAndSleep=set_mask_and_sleep
routine = someInterrupt1
routine = someInterrupt2
```

Example 1. ISRs in the SOP file

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_instance`
- `SynthOS_interrupt`
- `SynthOS_sleep`
- `SynthOS_start`
- `SynthOS_wait`

Note that `SynthOS_call` primitives can be inferred by SynthOS during synthesis. In these cases, the primitive is 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, ...);
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. 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 run in parallel by default. To run the task serialized, one at a time, the task should defined as `parallel = no` in the .SOP file for the project.

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.

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 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_check

Syntax:

```
flag = SynthOS_check(taskname);
flag = SynthOS_check(taskname[instance]);
```

Type: Non-blocking

Description:

This statement checks whether a Start task is currently executing. 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 start the same Start task, this primitive will return true if *any* instance of that Start task is scheduled for execution. If you would like to know if a specific instance of a Start task is executing, it is recommended that you invoke that Start 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).
`instance`	An optional parameter, an integer or an expression representing the instance number. It is relevant only for multi-instance Start tasks. If the value of Instance is greater than the maximum instance number, the system will call the panic routine.

Result:

`SynthOS_check` returns true if the task is executing, false if all instances of the task have completed execution.

SynthOS_instance

Syntax:

```
instance = SynthOS_instance();
```

Type: Non-blocking

Description:

This primitive returns the instance number for the currently running task. It can be used with multi-instance Loop task or Start task. In a multi-instance Loop or Start task, it is critical in some cases to know which instance is running. When this primitive is executed within a specific instance of a Loop or a Start task, it will return the instance number. The user can use it to set the behavior of the specific instance.

Parameters:

`instance`	The return value is the number of the currently running instance. This primitive is relevant for multi-instance Start and Loop tasks.

Result:

The return value is the instance number of the current running instance. This primitive is relevant only for multi-instance Loop task or Start task, and it can be used to manage the specific instance behavior.

Restrictions:

`SynthOS_instance` can be used only for multi-instance Loop tasks and Start tasks. Using it in any other case will generate a SynthOS error.

SynthOS_interrupt

Syntax:

```
SynthOS_interrupt(taskname);
```

Type: Non-blocking

This primitive is used to schedule an Interrupt task; this task will be executed immediately after the interrupt service routine (ISR) exits. It will not block the ISR execution.

Description:

This statement is used in ISR to schedule an Interrupt task, which will run immediately once the ISR exits and before the system returns to the previous task. This enables a high priority task to run (the Interrupt task) while enabling other interrupts to be triggered. Running the system with interrupts enabled ensures that critical and time-sensitive interrupts will still be handled. Calling `SynthOS_interrupt()` outside an interrupt service routine will result in the immediate execution of the Interrupt task.

Parameters:

`taskname`	This is the name of the task being called.

Restrictions:

An Interrupt task called by `SynthOS_interrupt` cannot include any blocking SynthOS primitives.

Any particular Interrupt task cannot be run multiple times concurrently. If a specific Interrupt task is called more than once, it will run only once after the ISR exits. If different Interrupt tasks are called, they will be executed in the order called (serialized) after the ISR is terminated. If an Interrupt task is called within any other routine other than an ISR, the scheduler will switch context and the Interrupt task will be executed immediately.

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, ...));
    SynthOS_start(taskname[instance](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.

Parameters:

`taskname`	This is the name of the task being called.
`a, b, c, ...`	These are the parameters for the called task.
`instance`	An optional parameter used for multi-instance Start tasks, this parameter defines the number of the instance to be started.

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:

An instance number larger than the maxInstances number defined in the SOP file will call the panic routine.

SynthOS_wait

Syntax:

```
  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 true.

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, global variables, or task names and optional task instances, such as `(taskname == 1)` or `(taskname[0] == taskname[1])`.

Parameters:

`condition`	This is any legal C expression such as (var1 * sin(x) < 5). The name of a task can be included in the condition, in which case the task name represents a Boolean value that is true when the task is idle and false when the task is executing. In cases of a multi-instance Loop or Start task, the condition should include the instance number in square brackets.

Result:

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 called as a start task.

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

It is not recommended to use volatile variables in the condition if they are changed in hardware rather than the software code. SynthOS tracks the variables in the condition and whenever one changes, the condition is reevaluated. When the condition includes a volatile variable that can be changed externally to the system—this includes I/O ports, memory and system registers—the system will not know these variables were changed, will not check for them, and the wait condition will not be reevaluated. See the Advanced Techniques section for information on monitoring volatile variables.

Using SynthOS

Synthesizing

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

```
SynthOS <sop_file>[options]
```

Example:

```
SynthOS project.sop
```

Options:

-D option	Pass *option* as a #define to the compiler for the generated code.
-h[elp]	Print an explanation of SynthOS user command line options and exit.
-I <path>	Add this *path* to the default search path for include files.
	If not specified, the default search path is the current directory.
-l <log *file*>	Send stdout and stderr output to *file*.
-o <directory>	Place the synthesized source code files in subdirectory *directory*. If not specified, the default subdirectory is SynthOS.
-v	Display the version number, date generated and the build number

Output:

SynthOS creates a synthesized version of each .c file along with several files implementing the synthesized operating system. The program entry point is a function `main()`.

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 of the started tasks and Loop tasks; when the scheduler is ready to schedule another task, it will always select the task with the highest priority that is ready to run (i.e., not blocked waiting for an event). If there are several tasks 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 SynthOS project file. When calling a task using `SynthOS_call()`, the task inherent the priority level from the calling task.

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 SynthOS project file for each task is set in the corresponding task section of the SynthOS project 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.

Comments

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

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 required for this project. It is specified by the keyword `version` followed by a version number.

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]
projectName = Project X
target = 68HC05
language = C
languageExtension = GCC
traceRoutine = debug_output
scheduler = roundRobin
contact = Vladimir Nabokov
company = PaleFire Corporation
website = www.palefire.com
email = vlad@palefire.com
```

projectName

This parameter defines the name of the user's project. This simply becomes a comment in the synthesized code for documentation purposes.

target

This parameter defines the processor that is to be targeted by SynthOS. This currently becomes a comment in the synthesized code for documentation purposes. In the future, it may change the synthesis process to take advantage of processor hardware resources.

processorSize

This parameter defines the bit-size (e.g. 8, 16, 32, 64) of the processor that is to be targeted by SynthOS. This currently becomes a comment in the synthesized code for documentation purposes. In the future, it may change the synthesis process to take advantage of processor hardware resources.

language

This parameter defines the computer programming language of the input source code and output source code. This currently becomes a comment in the synthesized code because C is the only programming language supported.

languageExtension

This parameter defines extension to the computer programming language of the input source code. Because different extensions to standard languages have slightly different syntax, this is required to parse the code correctly. The currently supported extensions are:

`None`	Default if no extension is specified

GCC	GNU C compiler
XC8	Microchip compiler

scheduler

This parameter defines the task-scheduling algorithm used for this project. The supported options are currently round robin and priority.

contact

This parameter defines the person responsible for the project. This simply becomes a comment in the synthesized code for documentation purposes.

company

This parameter defines name of the company that is creating the project. This simply becomes a comment in the synthesized code for documentation purposes.

email

This parameter defines the email for the person responsible for the project. This simply becomes a comment in the synthesized code for documentation purposes.

description

This parameter gives a multi-line description of the project. This simply becomes a comment in the synthesized code for documentation purposes.

traceRoutine

This parameter defines the function used for tracing the program execution and handling the debug information. This function receives the debug information from the system based on the different system trace level, which is defined by the user during the program compilation. The system has four trace levels. The higher the trace level, the more information is exposed to the user. The trace information can be handled in the trace function or sent to one of the I/O ports for the user to analyze.

Preprocessor section

The directives for the preprocessor are listed in the preprocessor section.

```
[preprocessor]
define = F_CPU=16000000UL
include = avr-include
include = avr-gcc-include
```

define

This parameter allows a compiler macro to be defined.

include

This parameter allows code to be included in a file during compilation.

Subproject Section

This specifies the subproject to be included in the main project. SynthOS merges the subproject into the project and generates the source files in a single directory.

```
[subProject]
file = ProjectName.sop
```

file

This the SOP file for the subproject to be included as part of the main project.

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.

Global interrupt section

This section defines global interrupt options for the system. The user then must fill in the specifics for each setting.

```
# The global interrupt routines are listed in the interruptGlobal section
[ interruptGlobal]
enable     = ON
getMask    = intGetIntMask
setMask    = setIntMask
enableAll = intEnableAll
setMaskAndSleep = sleepFunction
routine = someInterruptService1
routine = someInterruptService2
```

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

- `enableAll`: a routine that enables all interrupts. This is called before starting a loop/start/call/interrupt task or before returning control to a task after execution of a blocking primitive. This routine must have the signature: `void` *`routine`*`(void);`
- `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` *`routine`*`(void);`
- `setMask`: a routine that restores the interrupt status to the state specified in the parameter. The parameter value is normally the value returned by the getMask routine. This routine must have the signature: `void` *`routine`*`(int mask);`
- `setMaskAndSleep`: a routine that reduces power consumption by putting the processor into a wait state when the system is idle waiting for interrupts. This routine must have the signature: `void` *`routine`*`(void);`

Interrupts should be disabled until the `enableAll()` function enables them since interrupt handlers cannot work correctly when SynthOS is not initialized.

routine

This is the entry point for the interrupt service routine – ISR. ISRs need to be defined in the global interrupt section if you want to use any SynthOS services within the ISR including the `SynthOS_interrupt` primitive. You also need to specify the interrupt routine if it going to be use to get the system to return from idle loop.

Task section

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

```
# Task1
[task]
entry = Task1_routine
type = init
period = 1
priority = 1
```

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

# Task3
[task]
entry = Task3_routine
type = call
period = 2
priority = 3
parallel = yes
maxInstances = 2
maxRequests = 2
```

entry

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

type

This parameter specifies the task type.

The following types are valid.

call	Call task
idle	Idle task
init	Init task
interrupt	Interrupt task
loop	Loop task

period

This parameter specifies how many loops of the main polling loop in the task management code results 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 any other 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 any other scheduler has been specified.

maxRequests

This parameter is the maximum number of instances of the task that can be executing concurrently. This parameter only affects Start tasks; for Call tasks, this parameter is always 1.

parallel

This representswhether multiple instances of the Call task can run simultaneously (`parallel=yes`) or only one instance can run at a time (`parallel=no`). If left out, the default is `parallel=yes`.

maxInstances

This parameter defines the number of instances of the specific task that the system can support. In the case of a Loop task, the system will initiate and run the number of instances specified by `maxInstances`. In the case of a Start task, the user can spawn new instances of the task up to the number defined by this parameter.

Error handling section

The error handling section specifies how system errors are handled by the system.

```
[errorHandling]
panic = my_panic
```

Panic routine

This user-specified routine is called when the system encounters a problem from which it cannot recover. It gives the user an opportunity for a graceful shutdown.

One such error condition is when the system cannot allocate a spot in the execution queue for the task. Using the `maxRequests` parameter in the SOP file, the user can define for each Start task, the maximum number of requests that can be queued. At run time, the system will attempt to allocate a message in the queue for each time the task is started. If this number exceeds `maxRequests` and the system cannot allocate the memory, the system will call the panic function. The panic function takes as input a structure that includes the error code, the pointer to the task control block (TCB) of the calling task, and a pointer to the queue that failed. The panic routine can use this information for postmortem debugging. It is not recommended to return from this routine, but it can be used for the system to enter a graceful reset/shutdown process.

SynthOS_panic.h

A file with this name is automatically generated in the system code.

```
structSynthOS_Panic {
int panicCode;
    void *offender;
    void *location;
};
```

Example 2. SynthOS-generated panic routine code

The panic structure members are:

- `panicCode`: error code of the problem.
- `offender`: pointer to the currently running task that caused the failure.
- `location`: pointer to the object associated with the issue causing the system failure.

User code

Example 3 shows user code for implementing the panic routine.

```
void my_panic(struct SynthOS_Panic * panic)
{
     printf("Panic mode, code : %d\n", panic->panicCode);
     exit();
}
```

Example 3: User-defined panic routine

System Trace and Debug

Debug information in the generated code and the capability to trace the system status during execution are critical aspects of developing an embedded system, especially when it comes to real time applications. SynthOS can add system-level tracing to the code that allows the user to follow the system state during the program execution, and send debug messages through a predefined interface. The user can evaluate the system status and measure the system performance during its operation. This data can be logged for future use and analysis.

The system trace sends out messages regarding specific system level events. SynthOS has four levels of tracing available. The trace level is defined during compilation, enabling the user to change the trace level for an application without having to synthesize the code again. The trace levels for SynthOS are:

- Level 1: Task synchronization (SynthOS_wait execution)
- Level 2: Task execution (loop call and start of tasks)

- Level 3: Scheduler related events
- Level 4: Interrupt related events

Example 4 shows an example of a simple trace function. The parameters for the trace function are:

- Subject – the name of the task generating the message, in case of SynthOS internal message the subject will be "*system*"
- Verb – the activity being reported
- Object – the target of the reported operation

SOP

```
[project]
traceRoutine = my_trace
```

Code

```
void my_trace( const char * subject,
               const char * verb,
               const char * object)
{
     double long sys_time = clock();
     printf (log_file," %d %s: %s",sys_time, subject, verb);

     if (object == 0)
          printf("\n");
     else
          print("%s \n", object);
}
```

Example 4. Debug and trace information

There are four predefined levels of data tracing. Each level adds more information to the data tracing, with level 1 being the minimal trace information to enable the system to work at almost full speed. Level 4 generates the maximum data tracing available to help debugging system level problems. The tracing level is defined in the compilation phase of the development. This way you do not have to run SynthOS again to change the debugging level.

```
-D SynthOS_traceLevel=<trace level>
```

Level 1: Task lifecycle

- id100 - Current task waits on expression: the current task suspended its execution and waits for a specific expression to be satisfied.
- id101 -Task restarted on wait expression: the condition for a task that was holding on a wait expression was satisfied and the task was scheduled for execution.
- id102 -Entering Idle task (no active tasks): the ready task list is empty and there are no task pending for execution; the system enters Idle mode and will execute the specified Idle task

Level 2: Task activation of Call task and Start task

- id200 - Loop task started: the system Loop task started, after finishing the execution of all Init tasks.
- id201 - Loop task rollover: the Loop task finished execution and will restart (Loop task does not need a software loop in the code).
- id202 - Starting ("entered") Call task: the system execution moved to a Call task from the calling task.
- id203 - Call task terminated: Call task terminated its execution and will return control to the calling task.
- id204 - Start task scheduling ("request: a request for a Start task was placed in the queue; the task will be executed based on the priority and scheduling algorithm.
- id205 - Start task terminated ("finish: a Start task was terminated and taken off the ready queue.
- id206 - Start task message queue request empty ("suspended"): all instances of a specific task have terminated execution and there are no more pending instances of this task.

Level 3: Scheduler related and Interrupt task tracing messages

- id300 - Interrupt task terminated and the system returned to scheduler (“return to dispatcher”).
- id301 - Loop task or Start task released the CPU and returned to scheduler.
- id302 - The system returned to the Idle task after the run queue became empty.
- id303 - Start, Loop or Interrupt tasks are running.

Level 4: Interrupt task

- id400 - Interrupt service routine is called.
- id401 - Interrupt service routine returned.

Table 1describes the messages send by the system and the information transferred to the trace function. Note the information for the Verb variable is the actual text being sent to the function.

Trace level	ID	Description	Subject	Verb	Object
Level 1	100	Current task wait on expression	Task name	awaken by	Expression
	101	Task restarted on wait expression	Task name	Expression	
	102	Entering Idle task (no active tasks)	System	calling idle	
Level 2	200	Loop task started	Task name	entry	
	201	Loop task rollover	Task name	rollover	
	202	Starting (“entered”) Call task	Calling Task	Called task	
	203	Call task terminated	Task name	exited	
	204	Start task scheduling (“request")	Calling Task	request	
	205	Start task terminated (“finish”)	Task name	finished	
	206	Start Task message queue request empty (“suspended”)	Task name	suspended	
Level 3	300	Interrupt task terminated and returned to scheduler (“return to dispatcher”)	Task name	return to dispatcher	
	301	Loop or Start task release the CPU and “return to scheduler”	Task name	return to scheduler	
	302	The system returns to idle task after the run queue is empty	System	Idle task	
	304	Interrupt tasks are activated and run	Task name	runs	
Level 4	400	Interrupt task is called	Task name	being called	
	401	Interrupt task returned	Task name	returned	

Table 1: System trace messages and variables information

Coding Examples

A Simple Loop Task—“Hello, world”

Example 5 is 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.

SOP file

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

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

Code

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

Example 5: Simple Loop task

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

```
SynthOS helloWorld.sop
```

at the command line. SynthOS will create a subdirectory called `SynthOS` containing several different .c and .h files.

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 -o helloWorld.exe
```

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"

Example 6 is a slightly more complicated example with two tasks, one of which prints out "Hello" and the other prints out "world\n":

SOP file

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

[task]
entry = helloTask
type = loop

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

Code

```
// 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();
     }
}
```

Example 6: Two simple tasks

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 have 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.

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. Example 7is an example where the second task must not execute until the first task says it is 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");
    }
}
```

Example 7: Waiting on conditions

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

In some cases, you want to suspend a task until some external event occurs. There are many ways of accomplishing this, with the method of choice depending on the primitives offered by the operating system that you are running on. Example 8 is 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 notified when an event occurs.
void task() {

// 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() {

     // Wait for the event.
     SynthOS_wait(myEvent == 1);
     // Reset the event
     myEvent = 0;
     // 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;
}
```

Example 8: Synchronizing with ISRs

Loop and Idle Tasks

The Loop task in Example 9, called `loopTask`, spawns a Start task called `startTask` that runs in parallel with `loopTask`. At one point, `loopTask` pauses its execution and waits for `startTask` to terminate. The Start task waits for an external event, in this case a timer interrupt. At this point, none of the tasks are scheduled to run and the execution list is empty so that the system will start the Idle task, called `idleTask`, which will run in the background until it terminates or until an interrupt changes the system status. In the example, the Idle task will run until the `Timer` variable counts to 1000.

SOP file

```
[project]
idleRoutine = idleTask

[task]
entry = loopTask
type = loop

[task]
entry = startTask
type = start

[task]
entry = interruptTask
type = ISR

[interruptGlobal]
enable = on
```

Code

```
unsigned int Timer = 0;

void loopTask(void)
{
     SynthOS_start(startTask());
     // Do . . .
     SynthOS_wait(startTask);
     // Do . . .
}

void startTask(void)
{
     // Do . . .
     Timer = 0;
     SynthOS_wait(timer == 1000);
     // Do . . .
}

Void interruptTask(void)
{
     Timer++;
}
```

```
Void idleTask()
{
     // Set low power . . .
     // Do idle operation . . .
}
```

Example 9: Loop and Idle tasks

Advanced Programming Techniques

This section, for advanced users, gives programming techniques for getting extra optimization out of SynthOS.

SynthOS_wait() Condition Skipping

The condition used in `SynthOS_wait()` is evaluated at the end of the command and in condition statements (if, while, etc.). In the case when a variable is changed more than once while in the same command statement or condition, the condition may become true and then false before it is evaluated again. Example 10 shows a simple example of this problem.

Task1:

```
SynthOS_wait (v == 1);
```

Task2:

```
v = 1,v-2;    // <condition satisfied>, <condition not satisfied>
SynthOS_sleep();
```

Example 10: Limitation of SynthOS_wait()

In Example 10, the variable `v` is change twice within the same command statement. The condition is going to be evaluated only once the statement is complete and in this case it will always be false.

Task1:

```
SynthOS_wait(v == 1);
```

Task2:

```
v = 1;
```

Interrupt:

```
v = 2;
```

Example 11: Task order issues with SynthOS_wait()

In Example 11, we have a task and an interrupt that affect the value of variable `v`. The first task to be executed, Task1, sets the wait condition `v` to `1`. The next task to run, `Task2`, sets `v` to 1. This should satisfy the condition, but if an interrupt is asserted before the condition is evaluated, and the interrupt sets `v` to 2, the condition will not be satisfied once the execution returns to Task2. This means that the `Task1` wait condition will not be satisfied as long as Task 2 is not executed.

Monitoring Volatile Variables

The following are some ways to monitor volatile variables.

Continuous testing

Example 12 shows common way of checking for changes of a volatile variable is by continuous testing until the variable changes. During this time, the CPU will be occupied with this task. This is not efficient, but it can be a useful solution for short time events for a limited number of CPU cycles.

```
volatile reg;

for (i = 0; i < MAX_COUNT&& (reg & bit == 0); i++)
{
     // do stuff...
}
// The volatile reg variable has changed or we timed out of the loop
```

Example 12: Continuous checking of a volatile variable

Scheduler based testing

Example 13 shows a more efficient extension to the continuous testing method using `SynthOS_sleep()` to release the CPU and wait for the scheduler to return to the task later in the process. With this method, you cannot use the CPU sleep mode to save power in the system, but other tasks can continue to execute.

```
volatile reg;

while ((reg & bit) == 0)
{
     // do stuff...
     SynthOS_sleep();
}

// The volatile reg variable has changed
```

Example 13: Scheduled checking of a volatile variable

Interrupt Service Routine

Example 14 shows a method that is useful for inputs that can be monitored using hardware interrupts, like I/O pins, timers, and communication ports. You need to define a variable that is changed in the ISR. This variable can be monitored by the `SynthOS_wait()` statement. An advantage of this method is that you can utilize the CPU sleep mode to save power—the system will wake up on input changes.

```
anyTask()
{
     SynthOS_wait(trigger == 1);

     trigger = 0;
     // do stuff...
}

void ISR(void)
{
     trigger = 1;
}
```

Example 14: Checking volatile variable in ISR

Interrupt in Idle mode

This method shown in Example 15 is similar to the ISR method above, but uses the Idle task to manage the system while it is waiting for an external event to happen. In this method, all the running tasks suspend themselves using SynthOS_sleep or SynthOS_wait. At that point, the Idle task will start running as a background task until an interrupt is received and the system will execute the appropriate ISR. The ISR will signal to the tasks to resume their execution or can start a new task.

```
void anyTask(void)
{
     SynthOS_wait(trigger == 1);

     trigger = 0;
     // do stuff... }
}

Void interruptTask(void)
{
     Trigger = 1;
}
```

```
Void idleTask()
{
     // Set low power . . .
     // Do idle operation . . .
}
```

Example 15: running in idle task while waiting for an event

Parallel Tasks (Reentry Tasks)

Parallel task execution is the capability of the system to run a number of instances of the same task simultaneously. The system can re-enter the same task with different parameters and in a different context. All three basic task types can be run in this mode, as described below.

Call task

Call tasks can be called from Loop task or other call and started tasks. You cannot call a task from an Interrupt task or an Init task. Calling a task is a blocking primitive—it moves the execution thread to the called task, and the calling task is halted. Tasks that call a Call task arerun in parallel by default, meaning that if multiple tasksare running in parallel, each one of them can call the same Call task, and all those instances of the Call task will run concurrently. However, when a Call task is defined as a serialized task, by setting the directive `parallel = no` in the SOP file, it can be called by different tasks but only one instance of the task can be executed at any time. This means that all other tasks will wait on this queue for the previous Call task to exit. If the `parallel` directive is not defined for a specific task in the SOP file, the default behavior for the task is `parallel = yes`.

SOP file

```
[task]
entry = callTask
type = call
parallel = yes
```

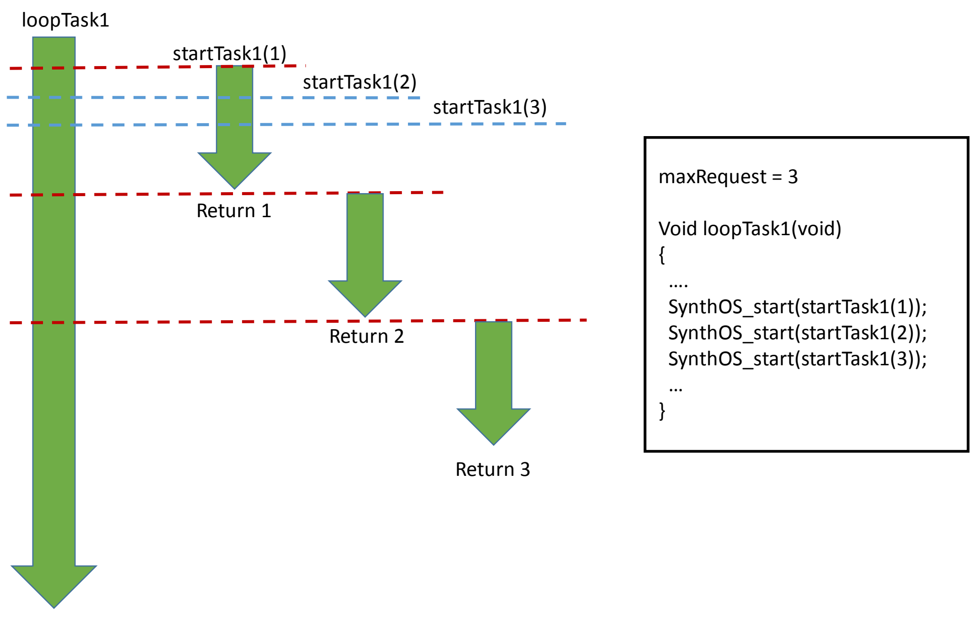
Code

```
void loopTask1(void)
{
     SynthOS_call(callTask(<variables>));
}

void LoopTask2(void)
{
     SynthOS_call(callTask(<variables>));
}

void  callTask(<variables>)
{
     // Do . . .
}
```

Example 16: Parallelized call tasks

Starting a multi-instance Call task

Tasks that are executed by using the Start task primitive run in parallel to the task that called them. You can start a number of different tasks and they will all run in parallel with the calling task as shown in Figure 2.



```
maxRequest = 3

Void loopTask1(void)
{
  ....
  SynthOS_start(startTask1(1));
  SynthOS_start(startTask1(2));
  SynthOS_start(startTask1(3));
  …
}
```

Figure 2: Execution of multiple instances of a Start task

If the user calls multiple instances of the same task, the task instances will run sequentially. To run the same task multiple times and have all instances run in parallel, you have to define the task as a multi-instance Start task and specify the maximum number of instances that can run in parallel. You can define a Start task as a multi-instance task by using the `maxInstances` parameter in the task configuration section of the SOP file. A good example is a device with multiple communication ports of the same type and protocol. The only thing that will change for the each communication task execution will be the port number. Using the task instance count, the task can get the configuration and data for a specific port. To be able to manage the multi-instances and know within a task which instance it is, the user can use the primitive `SynthOS_instance()`, which returns the instance number for the task that is running. This is shown in Example 16.

```
#define    COM1      0x00;
#define    COM2      0x01;

void loopTask(void)
{
     char data1[] = “this is com port 1”
     char data2[] = “this is com port 2”
     SynthOS_start( comTask [COM1] (baudRate, data1));
     SynthOS_start( comTask [COM2] (baudRate, data2));
}

void comTask(unsigned baudRate, ch *data)
{
     . . .
     send_data ( SynthOS_instance(), baudRate,Data);
     . . .
}
```

Example 17: Parallelized start tasks

As shown in Example 17, each one of this individual instance can also be called sequentially, but you have to adjust the `maxRequests` to match the number of call for the task. The system will put the start request into the task queue and execute them one at a time.

```
#define   COM1      0x00;

void loopTask(void)
{
     char data1[] = "this is com 1 first message"
     char data2[] = "this is com 1 second message"
     SynthOS_start ( comTask (baudRate, data1) );
     SynthOS_start ( comTask (baudRate, data2) );
}
```

Example 18: Serialized start task

Example 18 shows both options and the SOP file:

SOP file

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

[task]
entry = startTask
type = start
maxInstances = 2
maxRequests = 5
```

Code

```
void  loopTask(void)
{
     SynthOS_start(startTask [0](<variables>));
     SynthOS_start(startTask [1](<variables>));

     // This tasks will be executed serially
     // after the first task completes

     SynthOS_start(startTask [0](<variables>));
     SynthOS_start(startTask [1](<variables>));

}

void  startTask(<variables>)
{
     if(SynthOS_instance()== 0)
     {
          // do stuff...
     }
     else
     {
          if(SynthOS_instance()== 1)
          {
               // Do stuff...
          }
          else
          {
               // Do stuff...
          }
     }
}
```

Example 19: Combining serialized and parallelized Start tasks

Copyright Information

Index