Nucleus SE:
Quick Start Guide

Contents

Introduction

Nucleus SE is a small, simple, but complete real-time kernel. It has significant upward compatibility with the Nucleus RTOS family [like Nucleus RTOS] and finds application in some specific situations:

- Using a processor – typically 8- or 16- bit, but could be 32 – which is unsupported by Nucleus RTOS, particularly where memory footprint is a challenge.
- Using a supported processor, but with an application which cannot currently justify the cost and other overhead of a full-blown RTOS like Nucleus RTOS; Nucleus SE's upward compatibility may make its use attractive.
- In an educational situation, where the clarity and simplicity of code is an aid to learning; again the upward compatibility are attractive, along with source code distribution.

Nucleus SE offers a choice of 4 types of scheduler, which, along with other aspects of the system configuration, is selected at build time. The application program interface [API] is rich and is designed to implement a subset of the Nucleus RTOS API. Thus, applications and skills may be readily reused if a more powerful RTOS is required on a later occasion.

Nucleus SE supports the standard types of kernel objects: tasks, memory blocks, mailboxes, semaphores, event flags, signals, pipes, queues and timers. These are all created statically, their number being determined at build time.

A high level of scalability is offered, both on an API call basis, but also via kernel feature selection. The intention is to avoid the inclusion of any unwanted functionality and minimize memory footprint, particularly RAM – constant data is not copied, it remains in ROM.

Configuration

The simplest way to build an application using Nucleus SE is to make a copy of the **`src`** folder from the distribution [see *Distribution* below]. Then, add in the application program files and set up the Nucleus SE configuration files – the key ones being **`nuse_config.h`** and **`nuse_config.c`**.

nuse_config.h

This file contains a number of **`#define`** statements, which enable the user to configure the Nucleus SE build to their precise needs. The following settings are available:

- Number of each object type. Objects are static, so their number needs to be determined at build time. Nucleus SE supports between 1 and 16 tasks in an application. All other object types are optional and may each be instantiated 0 to 16 times.
- API call selection. Every API call may be enabled individually.
- Other system parameters. These include: scheduler type, support for task suspend, API parameter checking, support for initial task state, system time support, time slice duration, schedule count.

nuse_config.c

This file contains any required ROM data [initialized, of course] to support the objects enabled in **`nuse_config.h`**. RAM data space for objects that require it – partition memory, queues and pipes – may be defined here also.

Configuration Checking

It is the user's responsibility to configure Nucleus SE correctly for a particular application. Failure to do so will most likely be manifest in linker errors at build time. A header file – **`nuse_config_check.h`** – is included into **`nuse_config.c`** and performs a number of consistency checks on the settings in **`nuse_config.h`** [for example, enabling a mailbox API when no mailbox objects are being instantiated]. This yields **`#error`** statements which will appear during compilation and indicate the problem.

Schedulers

Nucleus SE supports 4 types of scheduler: Run to Completion, Round Robin, Time Slice, and Priority. The selection is made during configuration and composite scheduling mechanisms [e.g. priority with time slice] are not available.

Run to Completion [RTC]

This is the simplest scheduler. Each task is called in turn [respecting task suspend, if that is enabled]. Control is only returned to the scheduler when the task terminates [returns]. Each time a task is scheduled, it starts from the beginning, as no context is saved by the kernel. A "pseudo-context-save" may be implemented using static variables [and a **`switch`** statement] within a task.

Because this scheduler is almost trivially simple, it should be the first choice if it meets the needs of an application. Almost all the API calls may be used in usual way, except that those which suspend the task will immediately return. Since assembly language may be avoided entirely with the RTC scheduler, the code is very portable and easy to implement in a new environment. It can readily be used on a desktop computer, providing a simple prototyping or learning environment.

Round Robin [RR]

This is the more complex co-operative scheduler. Each task retains control until it explicitly relinquishes [with **`NUSE_Task_Relinquish()`**] or makes a call which results its own suspension [like **`NUSE_Task_Suspend()`** or **`NUSE_Task_Sleep()`**]. The next task in the sequence [respecting task suspend, if that is enabled] is then scheduled. Context is fully saved by the kernel.

Time Slice [TS]

This is a preemptive variant on the Round Robin scheduler. In addition to tasks being able to co-operatively relinquish control, the scheduler is called by the real time clock interrupt at a user-specified [in **`nuse_config.h`**] interval. This offers a quite straightforward way to ensure a fair distribution of processor time between tasks.

Priority

The priority scheduler allows up to 16 priority levels, with only one task at each level. Level 0 is the highest priority. Task priority levels are fixed at build time. When a task has control of the processor, it can relinquish it voluntarily if it makes a call which results its own suspension [like **`NUSE_Task_Suspend()`** or **`NUSE_Task_Sleep()`**] or if an interrupt results in a higher priority task being ready.

Stacks and Context

With the RTC scheduler, there is a single stack and no context saving.

With the other schedulers, each task has its own stack. The size and location of each stack is set individually [in **`nuse_config.c`**]. Full context is not saved on the stack [so no space needs to be allocated for it – only enough space for the application code and the needs of any native interrupt service routines]. Context is saved in a table of "context blocks" [**`NUSE_Task_Context[][]`**], which has one entry for each task; the size of an entry is processor dependent and set by the symbol **`NUSE_REGISTERS`** [in **`nuse_types.h`**]. The context loading and saving code is the only significant assembly language component of Nucleus SE.

Interrupts

Interrupt response and processing is key to real time software. It is essential that there is a minimal delay between an interrupt occurring and the processing code being executed. Furthermore, the code should spend as little time as possible in interrupt state, as this time is "stolen" from the scheduled tasks.

Nucleus PLUS has a sophisticated 2-level interrupt handling mechanism: HISRs and LISRs – see the Nucleus PLUS documentation for further details.

Nucleus SE takes a different, simpler approach: interrupts are either "native" or "managed".

A native interrupt is almost entirely unaffected by the fact that a kernel is in use. The interrupt service routine [ISR] may be constructed in the way defined by the development tools in use. Typically this requires the use of an extension keyword, **`interrupt`**, to ensure a basic context save. A pair of macros [**`NUSE_NISR_Enter()`** and **`NUSE_NISR_Exit()`**] should be inserted at the start and end of the ISR. Many API calls may be made from a native ISR, unless they can result in a context switch.

A managed interrupt is more complex [for non-RTC schedulers]. The ISR should be written as a regular function and its name passed to a macro: **`NUSE_MANAGED_ISR()`**. This results in a full context save and restore [using the Nucleus SE context blocks] at the start and end of the ISR. Almost all API calls can be made from a managed ISR.

The only ISR provided with the Nucleus SE distribution is for a tick timer [in **`nuse_real_time_clock_isr.c`**]. This managed ISR performs the following services:

- Process any configured timers; decrement counters and act upon expiration
- Increment the system clock, if configured
- Decrement any task timeout [sleep] counters and wake task as necessary
- If Time Slice scheduler, decrement counter and effect task swaps, as required

It is not essential for Nucleus SE to have a real time clock in order to run. However, certain services are obviously dependent upon such a facility.

Data

A user of Nucleus SE would normally communicate with the kernel exclusively via the API. Hence, an in-depth knowledge of the kernel data is not required. Notes here are included as an aid to understanding and to assist with porting.

Global Variables

Nucleus SE uses a small number of global variables, which are defined in **`nuse_globals.c`**:

`NUSE_Release_Info`
A character string indicating the Nucleus SE version.

`NUSE_Task_Active`
The index [see below] of the currently running task.

`NUSE_Task_State`
The current "state" – i.e. what kind of code is currently being executed; may be: **`NUSE_TASK`**, **`NUSE_STARTUP`**, **`NUSE_NISR`** or **`NUSE_MISR`**.

`NUSE_Task_Saved_State`
Previous value of **`NUSE_Task_State`** saved during an ISR.

NUSE_Task_Next
The index of the task to be run after an upcoming context switch; only with non-RTC schedulers.

NUSE_Tick_Clock
System clock value [only if configured].

NUSE_Time_Slice_Ticks
Time slice interval [only with TS scheduler].

#define Constants

Extensive use is made of symbolic constants in Nucleus SE. The **#define** mechanism is portable and well understood.

A large number of symbols are used to determine the configuration of a Nucleus SE build and these are set in **nuse_config.h**:

There are constants of the form **NUSE_object_NUMBER** [e.g. **NUSE_MAILBOX_NUMBER**] which determine how many instances of the object are required. They can be assigned values 0 to 16 [1 to 16 for tasks].

API calls are enabled by constants which are constructed by capitalizing the Nucleus SE API function name. For example, **NUSE_MAILBOX_SEND** enables the **NUSE_Mailbox_Send()** API call.

Remaining constants in **nuse_config.h** enable or control other aspects of Nucleus SE functionality:

NUSE_API_PARAMETER_CHECKING
This constant determines whether code is included to verify parameters on API calls. It is useful to enable this during debug, but maybe disable for production code to reduce memory footprint. Values may be **TRUE** or **FALSE**.

NUSE_INITIAL_TASK_STATE_SUPPORT
This constant enables the facility to specify initial task status. If enabled, a system may start with some or all tasks suspended; if disabled, all tasks are started with the status **NUSE_READY**. Values may be **TRUE** or **FALSE**.

NUSE_SUSPEND_ENABLE
This constant enables the facility whereby tasks may be suspended. If it is disabled, tasks are always running [**NUSE_READY**]. It must be enabled if the Priority scheduler is in use. Values may be **TRUE** or **FALSE**.

NUSE_BLOCKING_ENABLE
This constant enables the option of making blocking calls to relevant APIs – i.e. specifying **NUSE_TASK_SUSPEND**.

NUSE_SIGNAL_SUPPORT
As signals are not objects in their own right – they are “owned” by tasks – this constant determines whether they are included in a system or not [for all tasks]. Values may be **TRUE** or **FALSE**.

NUSE_TIMER_EXPIRATION_ROUTINE_SUPPORT
In Nucleus PLUS, when a timer expires, a function may be executed. This capability is optional in Nucleus SE and enabled by this constant. Values may be **TRUE** or **FALSE**.

NUSE_SYSTEM_TIME_SUPPORT
This constant enables the system clock and allows code to be included in the real time clock ISR to maintain it. Values may be **TRUE** or **FALSE**.

NUSE_SCHEDULER_TYPE
This constant determines which of the 4 available scheduler types is utilized by a Nucleus SE application. Its value may be one of the constants also defined here: **NUSE_RUN_TO_COMPLETION_SCHEDULER**, **NUSE_TIME_SLICE_SCHEDULER**, **NUSE_ROUND_ROBIN_SCHEDULER** or **NUSE_PRIORITY_SCHEDULER**.

NUSE_TIME_SLICE_TICKS
If the TS scheduler is in use, this constant specifies the time slice duration and it may have any value greater than 0; otherwise it must remain set to 0.

NUSE_SCHEDULE_COUNT_SUPPORT
Code to maintain a count of the number of times a task has been scheduled is generated, according to the setting of this constant. Values may be **TRUE** or **FALSE**.

NUSE_INCLUDE_EVERYTHING
This constant is for use during debugging and porting. It provides an easy way to enable all Nucleus SE optional facilities and all the API calls for instantiated objects. Values may be **TRUE** or **FALSE**.

A small number of symbols are defined in **nuse_types.h**. These are items that may be altered when Nucleus SE is being ported to a new processor or toolkit:

TRUE, **FALSE** and **NULL** are defined in fairly obvious ways to ensure consistency across toolkits.

ROM, **RAM** and **INTERRUPT** have blank definitions by default. They may be used to access language extensions in particular compilers.

NUSE_REGISTERS indicates the number of registers that need to be saved to preserve task context.

NUSE_STATUS_REGISTER is the value of a CPU status register which is placed on the stack prior to a return from exception in a context restore.

Object Indexes

In Nucleus PLUS, kernel objects are described by an object control block – a data structure held in RAM. As objects may be created and deleted dynamically, the location of these structures is potentially quite varied. The kernel keeps track of them via linked lists. An object is identified by a pointer to its control block.

Nucleus SE is implemented differently. Objects are static and are described by various arrays, which may be stored in RAM or ROM. An object is identified by its index into those arrays. If there are **n** instances of an object type, the indexes may have values from **0** to **n-1**.

With the Priority scheduler, the index of a task automatically implies its priority.

This alternative implementation serves to simplify the code, allowing the more limited addressing modes of some processors to be used. It also straightforwardly accommodates the distribution of data between ROM and RAM [with no unnecessary copying] and the optional inclusion of some features.

Kernel Object Data Structures

As mentioned previously, kernel objects in Nucleus SE are described by arrays in ROM and/or RAM. The existence of a particular array depends upon whether any instances of that object type have been configured and also, perhaps, whether a particular facility was enabled. The size of these arrays is determined by the number of instances of the object type.

The following arrays may exist in ROM [and are defined and statically initialized in **`nuse_config.c`**]:

`NUSE_Task_Start_Address[]`
The start address of the entry function for each task.

`NUSE_Task_Stack_Base[]`
The start address of the RAM area allocated to the stack for each task. This array does not exist if the RTC scheduler is selected.

`NUSE_Task_Stack_Size[]`
The size for each task. This array does not exist if the RTC scheduler is selected.

`NUSE_Task_Initial_State[]`
The initial state [**`NUSE_READY`** or **`NUSE_PURE_SUSPEND`**] for each task. This array only exists if support for initial task states is enabled.

`NUSE_Partition_Pool_Data_Address[]`
The address of the RAM area for storing the data for each partition pool.

`NUSE_Partition_Pool_Partition_Number[]`
The number of partitions in each pool.

`NUSE_Partition_Pool_Partition_Size[]`
The size of partitions in each pool.

`NUSE_Queue_Data[]`
The address of the RAM area for storing the data for each queue.

`NUSE_Queue_Size[]`
The size of each queue.

`NUSE_Pipe_Data[]`
The address of the RAM area for storing the data for each pipe.

`NUSE_Pipe_Size[]`
The size of each pipe.

`NUSE_Pipe_Message_Size[]`
The message size of each pipe.

`NUSE_Semaphore_Initial_Value[]`
The initial count value for each semaphore.

NUSE_Timer_Initial_Time[]
The initial timer count value for each timer.

NUSE_Timer_Reschedule_Time[]
The restart timer count value for each timer.

NUSE_Timer_Expiration_Routine_Address[]
The address of the timer expiration routine [function] for each timer. This array only exists if timer expiration routine support has been enabled.

NUSE_Timer_Expiration_Routine_Parameter[]
The parameter value for the timer expiration routine [function] for each timer. This array only exists if timer expiration routine support has been enabled.

The following arrays may exist in RAM [and are defined and dynamically initialized in **nuse_init.c**]:

NUSE_Task_Context[][]
The locations in which task context [the register set] is stored when a task is not in control of the processor. This array does not exist if the RTC scheduler is selected.

NUSE_Task_Signal_Flags[]
The locations in which the signal flags for each task are stored. This array only exists if signal support is enabled.

NUSE_Task_Timeout_Counter[]
The values of the sleep timeout counters for each task. This array only exists if the **NUSE_Task_Sleep()** API call is enabled.

NUSE_Task_Status[]
The locations where the status [**NUSE_READY**, **NUSE_PURE_SUSPEND** or suspend on an object code in the low nibble and the object index in the high nibble] of each task is stored. This array only exists if task suspend is enabled.

NUSE_Task_Blocking_Return[]
The locations where the return from suspend [blocking] status is located. This indicates success or reset of an object, as these are the only circumstances that result in a suspend being terminated. This array only exists if blocking API calls are enabled.

NUSE_Task_Schedule_Count[]
The locations in which the schedule counters for each task are stored. This array only exists is schedule counting is enabled.

NUSE_Partition_Pool_Partition_Used[]
The locations where the usage counters for each partition pool are stored.

NUSE_Partition_Pool_Blocking_Count[]
The locations where counts of how many tasks are waiting for partition pools are stored. This array only exists is schedule counting is enabled.

NUSE_Mailbox_Data[]
The locations where the data for each mailbox is stored.

NUSE_Mailbox_Status[]
The locations where the status [full/empty – **TRUE**/**FALSE**] is stored.

NUSE_Mailbox_Blocking_Count[]
The locations where counts of how many tasks are waiting for mailboxes are stored. This array only exists is schedule counting is enabled.

NUSE_Queue_Head[]
The locations where the indexes to the head of each queue are stored.

NUSE_Queue_Tail[]
The locations where the indexes to the tail of each queue are stored.

NUSE_Queue_Items[]
The locations where the entry count for each queue are stored.

NUSE_Queue_Blocking_Count[]
The locations where counts of how many tasks are waiting for queues are stored. This array only exists is schedule counting is enabled.

NUSE_Pipe_Head[]
The locations where the indexes to the head of each pipe are stored.

NUSE_Pipe_Tail[]
The locations where the indexes to the tail of each pipe are stored.

NUSE_Pipe_Items[]
The locations where the entry count for each pipe are stored.

NUSE_Pipe_Blocking_Count[]
The locations where counts of how many tasks are waiting for pipes are stored. This array only exists is schedule counting is enabled.

NUSE_Semaphore_Counter[]
The locations where the counters for each semaphore are stored.

NUSE_Semaphore_Blocking_Count[]
The locations where counts of how many tasks are waiting for semaphores are stored. This array only exists is schedule counting is enabled.

NUSE_Event_Group_Data[]
The locations where the data for each event group are stored.

NUSE_Event_Group_Blocking_Count[]
The locations where counts of how many tasks are waiting for event groups are stored. This array only exists is schedule counting is enabled.

NUSE_Timer_Status[]
The locations where the in-use status [**TRUE**/**FALSE**] of each timer are stored.

NUSE_Timer_Value[]
The locations where the current count-down value of each timer are stored.

NUSE_Timer_Expirations_Counter[]
The locations where counts are maintained of the expiration of each timer.

Distribution

Nucleus SE is supplied as source code, which may be deployed as required.

Folders

Expanding the ZIP file yields one file – this document – and a number of folders:

> **src** contains the source files for Nucleus SE. Details of the individual files may be found below.
>
> **simple demo** contains a basic example Nucleus SE application. There are two variants in sub-folders: **rtc** contains a version that will work with the Run To Completion scheduler; **priority** contains a version that will work with the Priority scheduler. In both cases a **readme** file gives details of the demo.
>
> **maximum dummy** contains an example Nucleus SE application, which is specifically designed to aid with porting or deployment on a new target. The files result in a configuration with a single [empty] task, a single instance of every kernel object type, all API calls enabled and every optional Nucleus SE facility enabled. There are two variants in sub-folders: **rtc** contains a version that will work with the Run To Completion scheduler; **rr** contains a version that will work with the Round Robin scheduler.
>
> **compatibility** contains files to enable maximum compatibility with Nucleus RTOS.

Files

Some notes about the files in the Nucleus SE distribution **src** folder:

The configuration files [**nuse_config.h**, **nuse_config.c** and **nuse_config_check.h**] were discussed under *Configuration* above.

There are a number of header files:

> **nuse.h** should be included in every application program file. It simply incorporates all of the header files below.
>
> **nuse_codes.h** contains definitions of all the symbolic parameters, error codes etc. used by Nucleus SE.
>
> **nuse_data.h** contains the **extern** declarations for all the global variables and kernel object data structures.
>
> **nuse_prototypes.h** contains the function prototypes for API service call, initialization and scheduler functions.
>
> **nuse_types.h** contains the **typedef**s for some standard data types used throughout Nucleus SE, along with a few symbolic constants. These may need to be revised if a change in toolkit or target processor is considered. Hence, this file also contains some very chip-specific symbols and macros that would also need attention.
>
> **nuse_binary.h** contains a series of **#define** constants that enable the easy specification of binary constants in the form **b00000000**, **b00000001** etc.

The API functions are grouped functionally in a number of files: **nuse_task.c**, **nuse_partition.c**, **nuse_mailbox.c**, **nuse_queue.c**, **nuse_pipe.c**,

nuse_semaphore.c, **nuse_event_group.c**, **nuse_signals.c**, **nuse_timer.c**, **nuse_clock.c**, **nuse_release_information.c**. The memory footprint is unaffected by the distribution of code across these files, as the build process does not rely upon the linker to make the selection. Rather, this process is realized using conditional compilation.

Processor independent scheduler code is contained in **nuse_scheduler.c**; the chip specific, assembly language components are in **nuse_context.c** [enclosed in **#pragma** directives – this enables conditional compilation to be utilized].

The entry point to Nucleus SE is **main()** in the file **nuse_main.c**. This calls the initialization code in **nuse_init.c** and then starts the task scheduler.

Global variables used by the kernel are declared in **nuse_globals.c**.

The only interrupt explicitly supported [if required] by Nucleus SE is a real time clock. The interrupt service routine is in **nuse_real_time_clock_isr.c**. The vector [and that for the context swap exception] is defined in **nuse_vector.c**.

Building a Nucleus SE Application

As mentioned above, in broad terms the simplest way to build an application using Nucleus SE is to make a copy of the **src** folder from the distribution. Then, add in the application program files and set up the Nucleus SE configuration files. See *Configuration* above.

The details vary slightly depending upon whether you are just building a demo [maybe to prove a port or just for education] or creating a new application.

Demo/Example

Having created the copy of **src**, follow these steps:

- Copy the files from the demo [in **simple demo** or **maximum dummy**] into your folder. This will overwrite some existing files.
- Copy the build script files [from **build**], into your folder.
- Run the build script.

Custom Application

Having created the copy of **src**, follow these steps:

- Add your application code source files to your folder.
- Modify **nuse_config.h** and **nuse_config.c** accordingly. See *Configuration* above.
- Copy the build script files [from **build**], into your folder.
- Modify the build scripts to accommodate your application files.
- Run the build script.

Porting Nucleus SE

To deploy Nucleus SE on a different target chip, or even move to a new software development toolkit, there are some issues to address. This is not intended to be a comprehensive porting manual, just a few guidelines:

- The file **`nuse_types.h`** needs to be reviewed, as it sets up the data types used by Nucleus SE. Additionally, some chip specific definitions and macros are included here.
- The other files containing assembly language – **`nuse_context.c`** and **`nuse_vector.c`** – will need to be reviewed and modified.
- The initialization code for tasks in **`nuse_init.c`** will need to be reviewed, but would not routinely need to be changed.
- Use the **`maximum dummy`** example builds to prove a new port, as these instantiate all Nucleus SE functionality and all code will be compiled.