Task Control Module For Kontakt 4/5

User's Guide

Nils Liberg
Robert Villwock

February 25, 2012

Table Of Contents

1.0 Introduction

The **Task Control Module**, TCM, is a recent enhancement to **Nils Liberg's KScript Editor** (the KSE). The TCM adds some long-sought-after KSP* extensions in the form of 'callable', parameterized functions and noteworthy multitasking support. An overview of the TCM is presented in section 1.1, followed by an overview and comparison of the KSE function types now available to Kontakt scripters in section 1.2. The remainder of this **User's Guide** covers everything you need to know to profitably use the TCM.

We strongly urge you to read this User's Guide in its entirety in order to get the most from all the services the TCM can provide for you. A **Technical Guide** is planned but in the meantime, if you are one of those users who want to tinker with the gears and pulleys under the hood, a pdf containing my original design guidelines is available on request.

The TCM was developed jointly by Nils Liberg and myself. I cooked up the design concept about a year ago but quickly realized using it with the then-existing KSE/KSP syntax would have resulted in very cryptic and error-prone coding. So, Nils developed the much-needed KSE support and integration and we both spent a lot of back and forth effort refining the initial design to where it is now.

We sincerely hope you will benefit from this latest upgrade to the KScript Editor. Without a doubt, the KSE remains the premiere tool for writing and maintaining Kontakt Scripts. If you haven't been using Nils' Editor, download and start using it today. You'll soon wonder how you ever got along without it.

* Kontakt Script Processor

1.1 Overview of the TCM

The TCM is a collection of routines and data structures that are integrated with the KSE to facilitate passing data to and from 'callable' functions. This new type of function has been given the name 'task function' and is defined with the keyword **taskfunc**. Task functions, **like** native KSP functions, are 'called' (as opposed to being expanded inline like standard KSE functions). But, **unlike** native KSP functions, task functions provide a standard parameter passing syntax. The current version of the TCM only supports integer parameters, ie simple variables, constants, or expressions reducing to a single integer. However the design of the TCM allows for the possibility of adding other data types such as arrays and strings. Whether or not such data types will be added in the future depends on perceived need and available time.

The TCM also provides some much-needed support for multitasking. Functions defined with **taskfunc** are inherently thread-safe and re-entrant. Moreover, task functions also support true local variables which are allocated on the active task stack and therefore are inherently thread-safe also. In addition, the TCM provides a number of thread-safe data storage and retrieval mechanisms including a new data type very similar to polyphonic variables (polyvars) but **useable in any callback**. See section 4.4 for a discussion of this new variable type.

Because the TCM is integrated with the new KSE compiler, all the gory details of parameter passing, and stack context switching are nicely hidden 'under the hood'. In fact, writing and referencing a **taskfunc** is no more difficult than writing and referencing the now familiar KSE functions. And, because the TCM is integrated with the KSE, you don't even need to import an external script module to use it. The TCM system software is essentially 'built into' the KSE, so simply including **tcm.init(max_tasks)** in your intialization callback (ICB) is all it takes to start using the TCM.

1.2 KSE Function Types

With the advent of the TCM, there are now 3 types of user functions available to the Kontakt scripter. The purpose of this section is to describe and compare these function types and to adopt some terminology that we can use throughout the rest of this User's Guide.

1.2.1 KS Functions

KS functions are the ‘original’ functions introduced with the first KScript Editor. Initially, Kontakt did not provide any kind of user function capability at all so KS functions were included with the KSE from its inception to provide this much-needed adjunct to the raw KSP language. KS functions provide the means for writing highly modular code that is far easier to write and maintain than the raw KSP syntax. Originally, KS functions provided no ‘return value’ (and thus no right-side assignment statement invokation) but they did (and still do) allow for output parameters to be passed. But, with the new compiler added in V1.4.8, the KSE now also includes user functions with a return value (and right-side reference functionality).

KS functions are very versatile in terms of where they can be used and the data types which can be passed as parameters. However, because KS functions have no native support in Kontakt, every invokation of a KS function results in the code body being inlined. So while KS functions provide a much-needed source code modularity, they do not provide the program-size efficiency of traditional ‘callable’ functions.

On the other hand, for smaller programs or where code-space efficiency is not important, KS functions are hard to beat. Because the code body is always inlined (and the parameters are essentially passed at compile time), KS functions have no ‘calling overhead’ to speak of. In fact, even with larger programs where code-space might be important, short functions are still probably best implemented as KS functions, especially when there are parameters to be passed. If there are no parameters to pass, then using a KN function (see section 1.2.2) might be a better choice if the routine is referenced multiple times in your code.

1.2.2 KN Functions

Since K4, the KSP now includes native user functions that can be used to write ‘callable’ code modules. We will refer to these native KSP functions as **KN** functions. KN functions can be written once and then invoked from multiple locations in your code (using the keyword **call**) without their code body having to be inlined (for each invokation) as with **KS** functions. Thus KN functions provide code-size efficiency in that the body of the function only has to be stored one place but can be invoked anywhere in your code (except from the ICB). On the downside however, KN functions do not provide for passing parameters so any data they manipulate must be handled by the scripter, usually via global variables.

The KSE provides a ‘pseudo-local’ variable type for use within a function body. These local variables are actually global variables with name variants assigned by the KSE compiler to prevent name conflicts when locals are named the same within several functions. Even though these pseudo-local variables are not true local variables, they can nonetheless provide many of the same benefits and they can be declared in both **KS** and **KN** functions. They can also be used within task functions (section 1.2.3) by using the explicit global/local directives as will be discussed in section 4.2.

1.2.3 Task Functions

With the TCM extension of the KSE, some of the best features of both **KS and KN** functions are now available with a new breed of function called a **task function**. While both KS and KN functions are declared with **function..end function**, task functions are declared with **taskfunc..end taskfunc**.

Task functions allow you to pass both input and output parameters and, optionally, to return a function value when referenced on the right side of an assignment statement. Of course, as with all generalized parameter passing protocols, these functions have some parameter-passing overhead connected with them. However, for lengthy functions they are very code-space efficient. And, if you need to write functions that are reentrant, task functions will probably be the best and most-efficient way to go.

Since task functions internally utilize KN functions at their core, they cannot be invoked from the ICB. However, you can always use the pgs trick to execute any initialization routines that need to reference either KN or task functions. For more detail about using task functions, see section 4.0

2.0 Using the TCM

2.1 KScript Editor Settings

To make the services of the TCM available to your script, you must use **Nils Liberg's KScript Editor**, V151 or higher (referred to throughout this guide as the **KSE**). You must also use the **new compiler** by **unchecking** *Use old compiler version* in the Settings menu. You should also enable the *Extra syntax checks* option. Due to some yet unexplained problem with initialization, whenever you switch from the old to the new compiler you will have to **close the KSE and then relaunch it**. This will insure that the KSE is properly initialized. You will only need to do this once as long as you stay with the new compiler. However, anytime you switch from the old compiler to the new, you will need to repeat this 'close and relaunch' to initialize the new compiler properly.

To produce more compact output code from the compiler (not only for the TCM system code but for your own application code as well), you should also enable the *Compact output* and *Optimize compiled code* options in the Settings menu. You may also check the *Compact variables* option to further 'densify' the compiled code and/or to make it less readable for 'prying eyes' if that has been your practice.

2.2 Initializing the TCM

To utilize all the services provided by the TCM, all you need do is invoke **tcm.init**(stack_depth) in your script's initialization callback (**ICB**).

```
tcm.init(35)   { this script requires a maximum stack depth of 35 words }
```

As shown above you must also specify the maximum stack size your application requires (see section 3.3).

3.0 Single and Multitasking

3.1 The Kontakt Model

KSP operation can be either single or multi-threaded depending on whether or not your script uses the **wait** function. Without waits, each callback will run to completion before any other callback will run. In this mode, callbacks are lined up in a queue and are only executed sequentially, thus operating as a single-task system. However, when a callback executes a **wait** statement, the callback is 'suspended' for a time and other callbacks are free to run before the suspended callback continues. Thus, callbacks containing waits can execute in an interleaved fashion similar to what is normally viewed as multitasking. So one way to view the Kontakt model is that callbacks represent independent processes or *tasks* and these tasks can execute sequentially or concurrently depending on whether or not the wait function is invoked.

However, since any callback that contains a **wait** statement has the potential to be reentered, this possible concurrency often gives rise to variables being overwritten unexpectedly. The problem can be illustrated easily with a wait-paced for loop. If your loop uses an ordinary KSP variable to hold the loop index, it will be **overwritten** if the callback is reentered during one of the loop's wait intervals. This will keep resetting the loop index so the loop will not execute as intended. The term *thread-safe* is often used to refer to variables (or other storage mechanisms) that are protected in some way from such reentrance problems.

The KSP only provides one limited form of thread-safe variable called a polyphonic variable (polyvar). Polyvars are essentially arrays indexed by the current note event's id. Thus, if you have a wait-paced for-loop that executes in the note or release callback (NCB/RCB), you can use a polyvar for the loop index and each concurrent execution of the callback will have its own copy of the loop index and therefore the loop will run properly. However, there is often a need to execute things like a wait-paced loop in other callback types, so polyvars alone do not solve the general problem of providing thread-safe data. What is needed is some kind of polyvar indexed by *callback* id rather than by *event* id and the TCM provides several such data types.

3.2 Thread Safety and tcm.wait

The TCM allows you to create multiple tasks that, like callbacks, can be executed concurrently. However, *unlike* callbacks, each TCM task has its own dedicated data structure known as a task *stack*. The TCM stack-based architecture is such that all TCM components are *inherently* thread-safe at the task level. And, by using a special variant of the **wait** function (known as **tcm.wait**), the concurrency of TCM tasks can be 'tied' to the concurrency of KSP callbacks. This allows the TCM to provide several thread-safe data types that are 'effectively' indexed by the *callback id*. Actually, TCM thread-safe data is indexed to the active *task id* but, if we 'tie' TCM tasks to KSP callbacks, the *task id* and *callback id* will relate to each other.

When **tcm.wait** executes, it behaves much like the simple **wait** function in that it suspends the current callback for the specified time (in usec) and allows other callbacks to run. However **tcm.wait** does more than that. For each **tcm.wait** executed, the TCM creates a new task with an empty stack (ready to support any new callback that might be triggered). Meanwhile, the current task is 'put to sleep' and it's stack is saved for later recall when the **tcm.wait** time expires. When that happens, the TCM restores its former stack to the exact state it was in when the task 'went to sleep'. Thus, the paused callback is free to continue without even being aware that anything happened during the wait period and all TCM thread-safe data will still be intact.

Since **tcm.wait** can be used anywhere that **wait** can be used (including within a function body), the easiest way to 'tie' TCM tasks to KSP callbacks is to replace all invokations of **wait** with **tcm.wait**. If you do this, task switching will always occur when a callback interrupts another callback and you will be able to use all TCM thread-safe data types without any further thought (see sections 4.2, 4.4, and 5.0) .

In certain special situations, you may be able to mix **wait** and **tcm.wait** but, it is not always obvious when you can and can't do this. When a callback is triggered during a simple **wait**, a new TCM task will **not** be created so the new callback will have to run in the same task as the suspended callback issuing the wait. There are situations in which this will not be a problem but such situations are usually tricky to analyze. However, advanced users that want to explore this possibility will find more information in section 7.0.

The penalty for using **tcm.wait** versus **wait** is minimal. In code space, **tcm.wait** compiles only one extra line of code for each invokation. Moreover, it takes only 0.5 microseconds or so to effect the task context switch. So, considering the rather imprecise pause time provided by **wait**, it is very doubtful that any performance difference will be noticed. Therefore, the only practical penalty for using **tcm.wait** will be a small reduction in cpu reserve for other things (and that only when the duty cycle of **tcm.wait** calls is high).

If your script uses no wait statements at all, then both the KSP and the TCM will operate in a single-task mode. When the TCM is first initialized (in the ICB), one task is activated with a single stack. As long as no **tcm.wait** function calls are made, all callbacks will be executed within that original task and no additional tasks will be created. Since the KSP also executes only one callback at a time, thread safety is no longer an issue so the TCM doesn't need to create more tasks nor provide thread-safe data structures. You can however still enjoy the basic feature of task functions as 'callable' functions that accept parameters. You can also use the task stack for scratch pad storage of temporary data, etc. For more on how to use task functions and the task stack, please see sections 4.0 and 5.0.

3.3 Determining the Maximum Stack Depth Needed

The stack is used to hold task function parameters and local variables (see section 4.0) plus any data that you might put on the stack temporarily (see section 5.0). What you need to do first is to count the total number of parameters plus local variables for each function defined with **taskfunc**. If any of the functions invoke other task functions (nested functions), start at the top level and then thread your way down through the nest, adding the parameters and locals at each level until you reach the lowest function in the nest.

Once you have thus determined the stack depth required for each top-level **taskfunc**, find the one that uses the most depth. That number plus any temporary data you intend to put on the stack is the maximum stack depth your application requires. However, it's always wise to add a little pad because you can easily make an error in this process or you may later write another function that becomes the new maximum-depth winner, etc.

On the other hand you don't want to set **stack_depth** unnecessarily large. The TCM allocates a fixed block of memory (currently 32K words) for the task stacks. However, since this block of **MEM_SIZE** must be divided between all the tasks, the larger you make stack_depth, the fewer the number of tasks that can be supported by the TCM. Therefore, once you have determined a generous (but not too generous) stack_depth for your application, you can estimate the maximum number of tasks that can be supported by dividing MEM_SIZE by stack_depth. You will find some guidelines in section 3.4 for how to determine if this is a sufficient number of tasks to support your application.

3.4 Estimating the number of Tasks Needed

To estimate the maximum number of tasks your application requires, you need to examine your code and find all the processes that invoke **tcm.wait**. Then, you need to determine the maximum number of times that each of these processes are expected to run concurrently in the worst case. The summation of all these possible concurrencies will be the maximum number of tasks needed.

Analyzing your script and determining the worst case concurrency is not always easy and the process can be somewhat error prone. However, if the estimate of MEM_SIZE/stack_depth (see section 3.3) indicates that a generously-larger number of tasks can be supported, everything should be cool. But, if your stack depth requirement is such that the number of tasks that can be supported are marginal (or worse yet, less than your needed task estimate), then you may have a problem. You will either have to limit some of the concurrency or rearrange your code so that maximum stack penetration will be reduced. However, you shouldn't worry too much about accurately estimating the maximum number of tasks that your application requires since the TCM will report an **exception** if your script tries to utilize too many tasks (see section 6.1). And, for most situations, 32K of total stack space should be more than adequate to support the number of tasks you will require (along with providing the maximum stack depth you require).

Finally, it should be mentioned that the actual number of tasks that the TCM will support for a given stack depth, is a little short what you would calculate with MEM_SIZE/stack_depth. The actual formula used by the TCM is MAX_TASKS = MEM_SIZE/STACK_SIZE - 1, but you can always verify the actual value assigned by reading the system constant MAX_TASKS (after you have executed **tcm.init** with your required stack_depth parameter).

3.5 KSP Inconsistencies

While it is true in general that any callback containing a wait can be reentered, there are some exceptions that you should be aware of. The **pgs callback** is apparently never retriggered until it actually exits. Therefore, pgs changes that occur while executing a wait in the pgs callback, will not retrigger the callback and moreover **such changes will not retrigger the callback at all!** Only changes that occur when the pgs callback is not running will actually trigger the callback. However, other callbacks are free to run while the pgs callback is in a **wait.**

The **ui_update** callback is really strange and it's behavior seems to depend on which particular ui element is clicked. In some situations when a **wait** is invoked from this callback, it will be aborted and the script may even be restarted (as if one had clicked the *apply* button). It is therefore recommended that you never use **wait** or **tcm.wait** in the **on ui_update** callback.

4.0 Writing Task Functions

Generally, Task functions are writen just like KS functions except that the body is defined between the keywords **taskfunc** and **end taskfunc** as illustrated below for a return-value taskfunc.

```
taskfunc sum_squares(x,y) -> result  { result = x^2 + y^2 }
 declare sq1
 declare sq2
 sq1 := x*x
 sq2 := y*y
 result := sq1 + sq2
end taskfunc
```

Note that local variables are written just like they are for KS functions but, in the case of task functions, these are 'true' local variables (as opposed to global variables with compiler-assigned name variations). Task functions are *invoked* the same way as KS functions are invoked. For example, to invoke **sum_squares** from the note callback (NCB), you would write the following.

```
on init
 q := sum_squares(3,4)  { the variable q will be set to 25 }
end on
```

A more detailed discussion of task function parameter passing and the use of local variables will be presented in sections 4.1 and 4.2 respectively.

4.1 Task Function Parameters

Task functions, like KS functions, can be written with or without a return value. The return-value format was illustrated in section 4.0. However, one-line return-value functions **cannot** be used in expressions like one-line KS functions can. The reason for this is that task functions require some prolog/epilog code which is generated by the compiler 'behind the scenes'. Therefore, if you were to write a one-line function, in reality it will actually be more than one line of code when compiled.

When task functions contain output parameters (as well as input parameters), the output parameters need to be *qualified* using the keyword **out** in front of the parameter and separated by a space. To illustrate this, let's recast the sum_squares function as a non-return value function.

```
taskfunc sum_squares(x,y,out sum)
 declare sq1
 declare sq2
 sq1 := x*x
 sq2 := y*y
 sum := sq1 + sq2
end taskfunc
```

In the above example, **sum** is *qualified* as an output parameter. Note that input parameters (that are only read by the function) do not need to be *qualified*. However, when parameters are used both as input *and* output (values that are altered by the function), you need to use the **var** qualifier. Suppose for example that we want to add the square of x and y and return the result in x.

We could write that this way.

```
taskfunc sum_squares(var x,y)
 declare sq1
 declare sq2
 sq1 := x*x
 sq2 := y*y
 x := sq1 + sq2
end taskfunc
```

As mentioned in section 1.0, the current version of the TCM only supports integer parameters. However, *input* parameters can be simple variables, constants, or expressions reducing to a single integer. To illustrate this using the first version of sum_squares, you could invoke it in the following ways.

```
q := sum_squares(3,4)   { q is set to 25 }
q := sum_squares(a,b)   { q is set to a^2 + b^2 }
```

or, if a = 5 and b = 6:

```
q := sum_squares(a+7, (b*a+9)/3)   { q will be set to 313 }
```

Note that **out** parameters (as well as **var** parameters) must be a single integer variable (not an expression or a constant). Specifically, parameters cannot be arrays or strings. But, as mentioned in section 1.1, the design of the TCM is extensible and data types such as arrays and strings could be added at a future date.

You can also use both **out** parameters *and* have a **return** value. For example you can write a task function like this:

```
taskfunc sum_and_sum_squares(x,y,out sum) -> result
 sum := x + y
 result := x*x + y*y
end taskfunc
```

Actually, the above example is equivalent to writing the following (except for how you obtain result).

```
taskfunc sum_and_squares(x,y,out sum,out result)
 sum := x + y
 result := x*x + y*y
end taskfunc
```

In the first case (the return-value function), the squares output is obtained by what you write outside of the function body.

```
ie      q := sum_and_squares(3,4,sum_of_xy)
```

while in the 2nd case, you are assigning the result within the function body (but the compiler then assigns it to **q** outside of the function body' but 'under the hood so that you aren't aware of it.

```
sum_and_squares(3,4,sum_of_xy,q)
```

You can also nest task functions just like you would KS functions except that you cannot use 'one-line-style' referencing *within* a function any more than you can *outside*. For example, if we rewrite the sum_squares function with nesting, we can write it as follows.

```
taskfunc sum_squares(x,y) -> result  { result = x^2 + y^2 }
  declare sq1
  declare sq2
  sq1 := Square(x)
  sq2 := Square(y)
  result := sq1 + sq2
end taskfunc

taskfunc Square(p) -> result
  result := p*p
end taskfunc
```

But, we **can't** write it this way.

```
taskfunc sum_squares(x,y) -> result  { This will be rejected by the compiler }
  result := Square(x) + Square(y)
end taskfunc
```

Finally, there is one nice feature of task functions that currently is not supported with KS functions. If you have a return value function that performs some action *and* returns a value, with KS functions you always have to provide a variable to receive the return value, even if you then throw it away.

But, with a return-value **taskfunc**, if you want the *action* of the function but have no need for the output value that is returned, you can simply write the function reference without supplying a variable to receive the return value.

To illustrate this, if you write the following as a KS function:

```
function ks.play_note(note,vel,ofst,mode) -> result
  result := play_note(note,vel,ofst,mode)
end function
```

Generally you would have to reference such a function as shown below, even if you don't need the return value.

```
id := ks.play_note(60,60,0,-1)
```

So, you may have to create a variable to use as a throw-away bucket (unless you already have a spare variable available for such a purpose).

However, when you write a similar **taskfunc**:

```
taskfunc tcm.play_note(note,vel,ofst,mode) -> result
  result := play_note(note,vel,ofst,mode)
end taskfunc
```

You can reference such a **taskfunc** either of the following two ways:

```
id := tcm.play_note(60,60,0,-1)   { if you want the id }
tcm.play_note(60,60,0,-1)         { if the id isn't needed. }
```

However, for such a short procedure, using a KS function would be much more efficient. The foregoing example was only presented to illustrate the return-value-throw-away option of task functions (not to suggest that you should always use a **taskfunc** for this kind of situation).

4.2 Local Variables

As illustrated in the prior sum_squares examples, local variables are declared in task functions the same way they are in KS functions. However, even though they are declared the same way, there are some noteworthy differences between task function local variables and the pseudo-local variables we have been using in KS functions.

Pseudo local variables declared within a KS function are actually global variables with name variations. As such, the next time such a function is invoked, the local vars will still contain the values they had when the function last exited. Task function local variables are however, 'true' local variables so their contents will be lost each time the declaring function exits. In other words, true local variables don't exist outside the scope of their defining function (see section 4.3).

On the other hand, because these variables are allocated on the active task stack, they are *inherently* thread safe. For example you can write the following task function:

```
taskfunc show_0_to_9
  declare n
  for n := 0 to 9
     message(tcm.task & '-' &n)    { Display each digit prefixed by the     }
     tcm.wait(1000000)             {  active task id for one second each   }
  end for
  message('')
end taskfunc
```

Now, if you invoke this function from the NCB, you can play any number of notes and each for-loop will run to completion. In other words, NCB reentrance will not *clobber* the for-loop variable because **n** is a true local variable (which is thread-safe). Actually a task will be created for each new note that occurs and each such task will have its own copy of n.

So, local variables declared in either KS or KN functions are pseudo locals while those declared in task functions are true local variables. However, if you have some kind of situation where you actually want to use pseudo-local variables (or for that matter an ordinary global variable), you can declare such variables within a task function by using the **local** and **global** directives to explicitly request these types. Note however that if you were to add the local or global directive to the for-loop variable **n**, the show_0_to_9 function would no longer be reentrant because **n** would not be thread safe.

4.3 Lifetime and Scope

Regarding **taskfunc** parameters and local variables, you need to understand the concepts of *lifetime* and *scope* so you will know when and where you can reference this data. The *formal* parameters of a function (the parameter names you use when you write the function, not the names of the *actual* parameters you pass when you invoke the function), as well as the local var names, can only be used within the body of the *defining* function. We will refer to this as the **scope** of the parameters. It means that you cannot reference the *formals* or *locals* outside of the function body nor can you access them from a different function even if that function is invoked within the defining function's body (ie invoked as a nested function). However, even though a caller's parameters cannot be referenced by the callee, when the callee exits (returns) to the caller, the caller's parameters can again be referenced by the caller. But, when the caller function itself exits, its *formal* parameters and *locals* are no longer accessible. Thus, the parameters of a function are only accessible by that function (this is their *scope*), and they no longer exist when the function exits (this is their *lifetime*). Advanced users may also want to read section 5.3 for a discussion of *lifetime* and *scope* for user stack data.

4.4 Task Variables

For use *within* a task function, local variables can be used for storing data that needs to be kept thread safe. But, within KS or KN functions (or for that matter any code that executes outside of any kind of function), we need another kind of thread-safe variable that we can use. Such variables are called *task variables*. Task variables are the TCM-counterpart of KSP polyvars. However, instead of creating arrays indexed by note-event id (like polyvars), task variables are indexed by the active task id. To illustrate the idea, suppose you want to create a thread safe variable named **xyz**. To create such a task variable, you would first declare an array named **xyz**[MAX_TASKS]. Using the system constant MAX_TASKS, insures that the array will be sized large enough to provide a copy of **xyz** for all possible concurrent tasks.

Then to access **xyz** as a task variable, you would write:

```
xyz[tcm.task]  { this accesses the copy of xyz dedicated to the currently active task }
```

The read-only property **tcm.task** always contains the currently active task id (which is an integer between 0 and MAX_TASKS - 1). Thus you can use **tcm.task** as an index into the **xyz** array so that each task will have its own unique copy of **xyz** (thereby keeping xyz thread safe at the task level).

To make creating and accessing such task variables easier, simply define a macro like the following:

```
macro declare_tvar(#name#)
  declare _#name#[MAX_TASKS]
  property #name#
    function get() -> result
      result := _#name#[tcm.task]
    end function
    function set(val)
      _#name#[tcm.task] := val
    end function
  end property
end macro
```

Then you can create as many task variables as you need for your application by simply writing something like **declare_tvar(abc)** in your ICB. And, once declared, you can reference such task variables as if they were simple variables. For example,

```
abc := 6        or     q := abc
```

Note that this makes task variables defined with **declare_tvar** almost the exact counterpart of polyvars. But, task variables are tied to the active task id rather than (the more-restrictive) active event id like polyvars are. Therefore, tvars can be used in any callback type to provide thread safety.

4.5 Task Function Limitations

Most of the limitations of task functions have already been mentioned but they are summarized here. One limitation is that a **taskfunc** cannot be invoked from the ICB. However, there should be little need to do such a thing anyway since task functions will usually be associated with concurrent task situations. But, if for some peculiar reason you do need to execute a task function as part of your initialization, you can always handle it as you would KN functions that need to be invoked from the ICB. Basically what you do is set a pgs variable at the end of your ICB. This will trigger a pgs callback right after the ICB exits. You can then test this pgs variable in the pgs callback (resetting it so this only happens once) and then execute a post-initialization routine. Since it is executed outside of the ICB, this post-iniitalization routine **can** contain **KN** and/or **taskfunc** invokations if needed.

Unlike KS functions with return values, you cannot use even a one-line **taskfunc** in expressions. When you invoke a task function on the right side of an assignment statement, it must be all by itself. This should pose little practical concern since one-line task functions would be very unusual anyway.

Task function parameters cannot be arrays, strings, families, or function names. In general task function parameters are assumed to be numerical integers (or something that reduces to a single integer). The reason for this restriction is that currently, all task function parameters are passed by value. However, the TCM design is such that passing parameters by reference could be implemented at some future date and if so, parameter types like arrays and/or strings could be added.

5.0 The TCM Stack

The TCM uses the stack for function parameters and local variables but you can also use the TCM stack at any time as a very convenient *scratch-pad area* for holding any kind of temporary data. For example to hold the interim results of various calculations or for passing parameters to and from KN functions, or more generally, for *parking* any data you want to keep thread safe. And if you follow a few simple rules which will be presented in section 5.2, you can use the stack easily and safely. However, you can also misuse the stack if you aren't careful so you should **always enable the DEBUG mode before doing any stack operations** (see section 6.3). With the debug mode enabled, the TCM will automatically detect and report any stack misuse and keep you from crashing the system with some illegal stack operation.

Finally, please keep in mind that everything that you can do with the stack can also be done with local variables and/or **tvars**. So, if you aren't perfectly comfortable with the following material, you can easily avoid using the TCM stack.

5.1 The Push/Pop Operations

A stack is an abstract data structure that functionally behaves in a manner very similar to a stack of objects such as dinner plates. When you want to add a plate to the stack, you place it on top of the existing stack and when you want to remove a plate, you take it off of the top of the stack. The stacks implemented in the TCM behave in a similar fashion so you can view a task stack as an array that can only be accessed on one end which is called the top. When we retireve data from the stack, we can only access the topmost item. When we store data in the stack, we always put it on top of the existing data already there.

Adding a new value to the top of the stack is usually referred to as 'a push' operation while removing a value from the top of the stack is referred to as 'a pop' operation. The TCM has two functions named **tcm.push** and **tcm.pop** to provide these operations. To illustrate how this pair of functions work, suppose we push the values 3, -9, and 25 onto the stack in that order as shown below.

```
tcm.push(3)
tcm.push(-9)
tcm.push(25)
```

Then, if we execute 3 pop operations like this:

```
x := tcm.pop() { x will be set to 25 }
y := tcm.pop() { y will be set to -9 }
z := tcm.pop() { z will be set to 3 }
```

When we finish the three pops, x, y, and z will contain 25, -9, and 3 as shown in the above comments. Since 25 was the last value pushed, it is the first value popped. Since 3 was the first value pushed, it is the last one popped. This last-in/first-out operation is why stacks are often referred to as a LIFO queue (or sometimes a FILO queue for first-in/last-out).

5.2 Using the Stack

To avoid problems, the stack should only be used with parity by always balancing your pushes and pops. Simply stated, **anything you push onto the stack during any process should be popped off before you exit that process**. For example, if you push data from the callback level itself (ie not within a **taskfunc**), you should also pop it off at the callback level **before exiting** the callback. If you push data from within the body of a **taskfunc**, you should pop it before you exit that function. If you have a nested **taskfunc** situation, any data you push while you are within a sub-level function should also be popped before you exit that sub-level function, etc.

Even more important, always push before you pop. That is **never try to pop data that you didn't previously push within the same process**. And, finally, enable **DEBUG** mode **before** you use the stack and until you are certain your code is free from stack misuse. However, the above guidelines are a little more restrictive than need be so if you want to use the stack more intellegently, you should read section 5.3 to gain a better understanding of what you can and can't do when using the TCM stack.

Unlike local variables which can only be used within a **taskfunc**, the stack (like **tvars**), can be used anywhere. That is, you can push/pop data at the top callback level, within a **taskfunc** or nest of taskfuncs or within the body of any **KN** or **KS** function or nest of functions. And, sometimes using the stack can save you having to create another **tvar** which of course needs to allocate another array. On the other hand, data placed on the stack is not named so you have to keep track of what you put where and you can only access the stack from the top so you have to order your pushes and pops to be sure you can 'get at' the data when you need it. In that regard, local variables and tvars are easier to use because you access the data by name and essentially have random access to it.

Finally, just to illustrate one simple way you could use the stack we'll recode the **show_0_to_9** taskfunc from section 4.2 as a KS function. A KS version of this function is shown below but keep in mind that this example is not very practical, it is only intended to illustrate the general idea.

```
function show_0_to_9
  declare n
  for n := 0 to 9
    message(tcm.task & '-' & n) }
    tcm.push(n)
    tcm.wait(1000000)
    n := tcm.pop()
  end for
end function
```

If this function is invoked from the NCB and we hit a bunch of keys, each key will initiate a separate loop. And, even though the loop index is only a pseudo-local variable (and thus not thread-safe), the loop will nonetheless run properly because we are using the stack to save and restore it before and after the pause.

Note that the above example would also work if **show_0_to_9** were 'called' as a **KN** function from the NCB. Or, for that matter, if we didn't use a function at all but rather coded the loop inline in the NCB with **n** declared as a global variable in the ICB. More importantly, it would also work in another callback type. For example if we were to code this in the callback handler for a switch, we could repeatedly depress the switch and each loop thus triggered would run correctly.

5.3 Stack Data Lifetime and Scope

If you want to use the TCM stack more intelligently, you need to understand the rules of *lifetime* and *scope* for stack data. If you are in a **taskfunc** body when you push data onto the stack, the *scope* and *lifetime* of the stack data is the same as the *scope* and *lifetime* of the parameters and local variables for the **taskfunc** that pushes it (see section 4.3). Thus such data can be popped back off only while you are still in the body of the same **taskfunc** and have not yet exited that function (since pushing the data). If you are not in a **taskfunc** body when you push data onto the stack, the *lifetime* and *scope* of the stack data is the same as the *scope* and *lifetime* of the task itself. Therefore you can pop the data back off anywhere within the same task provided the task has not been 'retired' (see section 7.0) since pushing the data.

The following example may help to clarify the above rules on *scope* and *lifetime* of user stack data. Consider the following note callback. The callback uses the stack and also invokes the two functions named **Parent** and **Child**. Each of these functions also use the stack and **Parent** invokes **Child** in a nest. Finally, the **Parent** function includes a 2-sec pause and the callback includes a 1-sec pause.

```
on init
    tcm.init(100)
    declare y1
    declare y2
    declare y3
    declare y4
end on

on note
    tcm.push(23)
    tcm.push(-64)
    Parent
    y1 := tcm.pop()    { y1 will contain -64 }
    Child
    tcm.wait(1000000)
    y2 := tcm.pop()    { y2 will contain 23 }
end on

taskfunc Parent
    tcm.push(5)
    tcm.push(-7)
    tcm.wait(2000000)
    Child
    y3 := tcm.pop()    { y3 will contain -7 }
end taskfunc

taskfunc Child
    tcm.push(1234)
    tcm.push(5678)
    y4 := tcm.pop()    { y4 will contain 5678 }
end taskfunc
```

If we hit a key, the NCB will be triggered and the following will happen. First the values 23 and -64 are pushed on the stack and then the **Parent** taskfunc is invoked. This function begins by pushing the values 5 and -7 onto the stack and then pausing for 2-seconds. During this pause other callbacks may run or be continued but when the pause expires, Parent continues as if nothing had happened.

After the pause, Parent invokes Child which in turn pushes 1234 and 5678 onto the stack and then pops 5678 right back off to the global variable named y4. Note that when Child exits, it has left 1234 on the stack. Now, when Parent resumes (after Child *returns*) it pops the stack to y3. Now you might think that the value popped will be the 1234 (that Child left on the stack) but you would be wrong on two counts. First, the *scope* rules dictate that the 1234 cannot be accessed by any **taskfunc** other than Child. Secondly, the *lifetime* rules stipulate that Child's stack data will be lost when Child exits. Therefore, what is popped to y3 is the last value that Parent pushed onto the stack, namely -7.

After popping to y3, Parent *returns* to the NCB where the stack is popped to y1. Note that Parent has left the value 5 on the stack when it exits. So, again we might be tempted to think that the value popped to y1 will be this *leftover* 5. However, *scope* and *lifetime* rules tell us that the leftover 5 will be lost and the value popped to y1 will be the last value pushed onto the stack at the callback level, namely -64.

Next, the NCB invokes the Child function again and since Child previously left 1234 on the stack (when it returned to Parent) you might think that the stack for Child still contains the leftover 1234 and now that we are *in scope* again that we could access it when we enter Child for this second time. But, while we are in the right *scope*, we are in the wrong *lifetime*. Since Child has already exited from the invokation by Parent, the first 1234 that Child pushed has been lost. Continuing now with this second invokation of Child, first the values 1234 and 5678 are pushed onto Child's *empty* stack and then the 5678 is popped back off to y4. When Child then returns to the NCB, it again has left the value 1234 on the stack.

Upon return of Child to the NCB, the NCB then requests a 1-sec pause afterwhich it pops the stack to y2. Again the value popped will not be the leftover 1234 but rather the last value pushed on the stack at the callback level, namely 23. The NCB now exits with nothing left on the stack.

Normally, you wouldn't leave data on the stack like we did with this example. When this is done at the function level, excess data is simply discarded as we have seen with the example presented. However, if we leave data on the stack at the callback level, it will stay there until the corresponding task is **retired** and it's not always easy to determine just when that will occur (see section 7.0). Therefore, unbalanced pushes at the task level can cause stack creep and eventual stack overflow. Worse than stack *overflow* however is stack *underflow* which in some cases can crash the system. Therefore never pop data you haven't previously pushed at the same process level. To be sure of this, it is best to activate the DEBUG mode (see section 6.3). Debug mode will detect and report both stack over and underflow and will keep you from inadvertently crashing the system during your script's development.

6.0 TCM Exceptions

When an exception occurs, the TCM will trigger the pgs callback and set **tcm.exception** (a read-only variable) to one of several possible error codes. The kind of exceptions that the TCM can detect and report (and how your script can use this information during development and/or to notify users of your script) will be discussed in the following sections.

6.1 Task Overflow

If the maximum concurrency of your application ever tries to exceed the maximum number of tasks that can be supported by the TCM (see section 3.4), an exception will be created with **tcm.exception** set to the value of a system constant named **TOO_MANY_TASKS**. This task overflow condition will always be detected and reported regardless of whether the **DEBUG** option (see section 6.3) is on or off.

6.2 Stack Overflow/Underflow

Stack overflow can occur whenever one of your most stack-hungry tasks tries to execute too deep a nest of task functions and/or when you have pushed too much temporary data on the stack (see section 3.3). It can also happen due to stack creep if you use more pushes than pops at the callback level (see section 5.3). Stack underflow can occur whenever you inadvertently try to pop more data than you previously pushed onto the stack at the same level.

When **DEBUG** mode is activated (see section 6.3), it continuously monitors for illegal stack operations that might cause stack overflow or underflow and will report an exception if such an illegal condition occurs. Moreover, since illegal pops can corrupt the TCM itself, the DEBUG system will not only report such an underflow, it will also prevent it from crashing the system.

Therefore, during development of your script, you should always enable the DEBUG mode. However, the monitoring of stack operations does add some overhead to the TCM's operation so after you are confident that your script is free from illegal stack operations, you can disable the DEBUG system. But, if you are not trying to push your cpu to the limit, you can choose to leave DEBUG active all the time if you wish. Please see section 6.3 for how to enable the DEBUG mode.

6.3 Using Debug Mode

The debug mode can be activated by including **SET_CONDITION(TCM_DEBUG)** in your script's ICB. As mentioned in section 6.2, the TCM can detect and report stack overflow or underflow with the debug mode enabled. Therefore, it is recommended that you activate debug mode during development of your script. After you are reasonably confident that you don't have any potential stack problems you can recompile your script *without* the SET_CONDITION statement. Alternatively, you can also choose to leave debug mode active all the time if you don't mind a little overhead code and cpu usage.

6.4 Handling Exceptions

As already explained in section 6.0, the TCM reports exceptions by triggering the pgs callback and setting **tcm.exception** to an appropriate error code. When there are no exceptions, **tcm.exception** will be zero. However, if an exception occurs, the value stored in **tcm.exception** will be one of several non-zero error codes available as system constants with the following names:

TOO_MANY_TASKS
STACK_OVERFLOW
and **STACK_UNDERFLOW**

Therefore, you can write an KS exception handler along the lines shown below and invoke it from your pgs callback.

```
function exception_handler
  select tcm.exception
    case TOO_MANY_TASKS
      message('Too many tasks')
    case STACK_OVERFLOW
      message('Stack overflow')
    case STACK_UNDERFLOW
      message('Stack underflow')
  end select
end function
```

For the above illustration, the handler only outputs messages to the Kontakt status line. However, for your actual handler, you will probably want to display something a little more noticeable for each case. Further, when an exception occurs, your script may cease to work properly and loud notes could be left hanging so, it might be a good idea to include something like **note_off(ALL_EVENTS)** in your specific handlers.

Also, since the pgs callback can miss triggers that occur while the callback is still running (see section 3.5), you should probably position your exception_handler at the very end of the pgs callback (or at least after any pauses). That way if a TCM exception occurs while the pgs callback is paused, your handler will still notice the exception before exiting the callback. Exceptions that occur *after* the pgs callback exits will of course retrigger the pgs callback and thus be noticed.

7.0 Relating Tasks to Callbacks

As discussed in section 3.2, when you replace all **wait** calls with **tcm.wait**, you effectively 'tie' task concurrency with callback concurrency. However, if you want to mix **wait** calls with **tcm.wait** calls, it will be important for you to understand just how the TCM relates tasks to callbacks. Only then will you be able to determine just when and where you might be able to use **wait** instead of **tcm.wait**. For sophisticated users who want to explore this possibility, the following summary of how the TCM operates may prove useful.

TCM tasks have three possible states which can be defined as **retired**, **active**, and **suspended**. Initially all but one task is marked as *retired*. The one task not marked *retired* is marked as *active*. This *active* task is then *standing by* to support callbacks which may invoke task functions and/or use the stack. The *active* task remains unchanged until the first callback containing a **tcm.wait** gets triggered. However, when the first **tcm.wait** request is encountered, the *callback id* (as well as the *active* task's stack state) is saved and a new *active* task with an empty stack is created. The saved task is then marked as *suspended* and the new *active* task is now *standing by* to support any new callbacks that may be triggered during the suspension.

When one or more tasks are *suspended*, eventually one of the paused callbacks will time out and the TCM will regain control. The TCM will then determine which callback has resumed (by reading its *callback id*), and then restore the saved state for that callback. The TCM will then *retire* the current 'stand-by task' task (thus ending its *lifetime*) and the resumed task then becomes the *active* task.

From the foregoing you can see that there is **always** an *active* task standing by but there is **never more than one** *active* task at a time. All other tasks are either *suspended* or in the available task pool (ie *retired*).

8.0 Reserved Names and System Constants

Since Kontakt doesn't provide any support for keeping variable and function names private, there is always the possibility that you might accidentally use a name in your script that the TCM is already using. Of course the KSE will warn you if you declare a name that's already in use but, nothing prevents you from referencing one of the system functions or variables. There is little harm as long as all you are doing is reading a system variable but, if you execute a system function or alter one or more system variables you could end up crashing your application.

The purpose of this section is to list the names used by the TCM and tell you which ones are in the 'off limits' category. Consulting this list may save you from getting compiler errors due to name duplication, but more importantly, it can help you to avoid clobbering a critical system variable unintentionally.

Variables that you need to read but aren't allowed to alter are coded as read-only properties so you won't be able to accidentally overwrite these values using the same name. System constants can of course only be read and not altered so these don't need to be protected.

User Interface

Functions		Read-Only Variables	
tcm.init	**tcm.push**	**tcm.exception**	{ exception code }
tcm.pop	**tcm.wait**	**tcm.task**	{ active task id }

Compiler Switch

TCM_DEBUG { use **SET_CONDITION(TCM_DEBUG)** to enable debug mode }

System Constants

TOO_MANY_TASKS	{ exception codes }
STACK_OVERFLOW	
STACK_UNDERFLOW	
MAX_TASKS	{ maximum tasks supported }
MEM_SIZE	{ available memory for all task stacks }
STACK_SIZE	{ maximum stack depth that you specified }
TASK_0	{ an internal value used by the TCM }

System Off Limits

Variables		Functions
fp	**tstate.fp**	**check_empty**
p	**tstate.fs**	**check_full**
sp	**tstate.fs**	**prolog_end**
tx	**tstate.sp**	**set_exception**
TCM_EXCEPTION { pgs key }		**_twait**

If you avoid referencing the 9 variables and 5 functions listed above as 'System Off Limits', you should have no problems with your script using the TCM system.