Motion Control Products

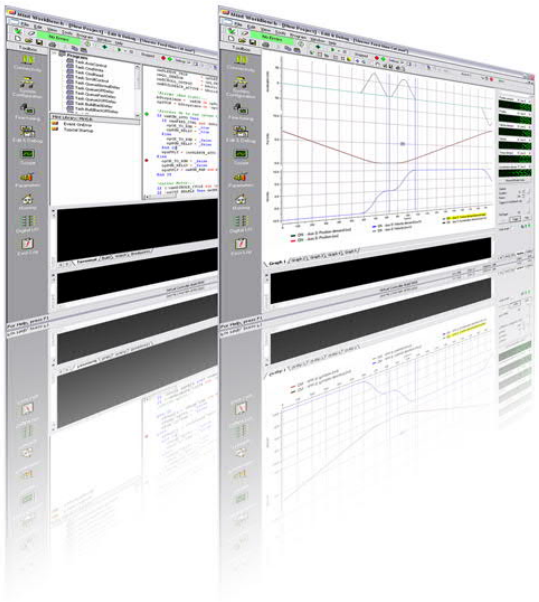
# Application note
# Multi-tasking basics

AN00107

Rev C (EN)

MintMT introduces the ability to easily write programs that contain multiple tasks. This is a powerful tool, allowing the development of code for controlling multiple axes without the worry of manually interlacing the commands for each axis. Tasks are not limited to motion control and can be used for any purpose, such as PLC style logic control and/or a user interface.

The ability to individually adjust the priority of each task allows the user to maintain the desired responsiveness for each task. For example, an HMI task may be given a relatively low priority to ensure that it does not hinder the more important motion control tasks.



## Introduction

Tasks are declared using the Task and End Task keywords. These keywords enclose the statements that will be executed when the task is run and also allow the name of the task to be specified. Note that in MintMT, each object, whether it be a variable, subroutine, define, etc., or even a task, must have a unique name. An example of a simple task declaration is shown below.

```
Task myTask
  Print #1 "INX.0 = ", INX(0)
End  Task
```

Executing a Task

In MintMT, only the parent task executes initially by default, all other tasks start in a dormant state. Effectively, the parent process is in charge of starting any tasks that are required to run. This is achieved using the Run command, which may take the name or names of the tasks required to be started.

```
Run myTask

Task myTask
  Print #1 "INX.0 = ", INX(0)
End Task
```

Note that there is no need for an explicit End statement after the Run command, as execution is implicitly terminated after the last of the parent tasks commands. This is to ensure that execution never 'spills' into any tasks, subroutines, functions or events defined below.

### Waiting for a Task to Terminate

The problem with the previous program is that the task starts, but immediately stops! “Why?” you might ask, and the reason for this is that as soon as the parent task terminates, all child tasks also terminate. To get around this, the parent task must be made to wait using the Pause command, which waits until the expression that follows it becomes true. The best way to wait for a task to terminate is to monitor its status using the TaskStatus function. A task will always be in any one of the following states:
_tskTerminated
_tskRunning
_tskSuspended
_tskWaiting
Our example program then becomes:

```
Run myTask
Pause TaskStatus(myTask) = _tskTerminated

Task myTask
  Print #1 “INX.0 = ”, INX(0)
End Task
```

### Making a Task Continuous

Our example program is not very useful in its current form, because it just prints the value of a digital input once before terminating. This illustrates the point that tasks run until their last statement, (the one before the End Task delimiter) and then terminate. This behavior may be what is desired, but in this case we want to print the value every 500ms continuously.

```
Run myTask
Pause TaskStatus(myTask) = _tskTerminated

Task myTask
  Loop
    Print #1 “INX.0 = ”, INX(0)
    Wait = 500
  End Loop
End Task
```

The Pause statement is now a bit redundant, at least in its present form, and could safely be replaced by “Pause _false”. Instead though, we’ll retain this form of termination, but add another task for the HMI.

### Controlling Execution Using an HMI Task

Typically, an application will have a number of tasks whose primary role is machine control and another task providing the human/machine interface for example. This HMI task could be a child task, or the parent task could be used for this purpose. Whichever means is chosen is up to personal preference, but having a child task allows the HMI specific data to be hidden away inside a task. This may not seem advantageous in the program below, but for a more complex user interface, that will typically employ a menu, and possibly sub-menus, it does have its benefits.

```
Run myTask, hmiTask
Pause TaskStatus(hmiTask) = _tskTerminated

Task myTask
  Loop
    Print #1 “INX.0 = ”, INX(0)
    Wait = 500
  End Loop
End Task
```

```
Task hmiTask
  Print “Press ‘X’ to exit”
  Pause InKey(1) = ‘X’
End Task
```

Note that in the above program, we only have to monitor the state of the HMI task, as the other task always runs continuously.

### Communicating Between Tasks

Sometimes it is necessary for one task to share information with another task, and this can be achieved using a global variable. Global variables are declared in the parent task and are visible in all modules, irrespective of whether they are tasks, events, subroutines, functions or the startup block. If we change the HMI to allow the adjustment of the input bit to monitor while the program is running, it should illustrate how this operates.
Note: It is important to avoid having two tasks that both alter the value of the same variable. If this occurs, then you may not get the result that you thought you would due to the time slicing used by the scheduler to multi-task and the way that the instructions get interleaved.

```
Dim nBit As Integer = 0

Run myTask, hmiTask
Pause TaskStatus(hmiTask) = _tskTerminated

Task myTask
  Loop
    Print #1 “INX.”, nBit, “ = ”, INX(nBit)
    Wait = 500
  End Loop
End Task

Task hmiTask
  Print “Press a digit to set the bit to monitor”
  Print “Or ‘X’ to exit”
  Loop
    Pause InKey(1)
    Select Case LastKey
      Case '0' To '7'
        nBit = LastKey - '0'
      Case 'X'
        Exit Task
    End Select
  End Loop
End Task
```

The above program has the benefit of being small, so it is reasonably clear what variable nBit does. However, with a larger program it would not be as readily apparent what all the variables did, and this problem increases the larger the program becomes. In order to avoid the proliferation of global data, the block structuring in MintMT allows data to be declared inside a module, which is beneficial because it hides the data from other modules, ensuring that it cannot be inadvertently used elsewhere.

You might be wondering what use this is when it comes to sharing data between tasks, but there is a way to make this local ‘hidden’ data visible by using the scope override operator (::). This operator allows variables to be qualified by the module they reside in by specifying the name of the module ahead of the variables name, but separated by a double-colon. The program below shows how this is done.

```
Run myTask, hmiTask
Pause TaskStatus(hmiTask) = _tskTerminated
```

```
Task myTask
  Dim nBit As Integer = 0
  Loop
    Print "INX.", nBit, " = ", INX(nBit)
    Wait = 500
  End Loop
End Task

Task hmiTask
  Print "Press a digit to set the bit to monitor"
  Print "Or 'X' to exit"
  Loop
    Pause InKey(1)
    Select Case LastKey
      Case '0' To '7'
'* Access the 'nBit' variable that belongs to 'myTask' using the override operator (::)
        myTask::nBit = LastKey - '0'
      Case 'X'
        Exit Task
    End Select
  End Loop
End Task
```

Note: The scope override operator (::) can only be used to access data in modules of class Task, Event and Startup, and not in modules of class Sub and Function.

### Altering the Priority of Tasks

By default, tasks always start with a priority of ten, which is suitable for most applications. However, there are instances where it may be imperative that a task executes more rapidly than other tasks. Conversely, it may be suitable for another task to run much more slowly than other tasks. An example of a high priority task would be one that controls motion, and a low priority task might be an HMI.

Our example program does not really gain anything from adjusting the priorities of its two tasks, but to illustrate how it is performed. The program below sets the HMI task to have a low priority and the input-monitoring task to have a high priority.

```
TaskPriority myTask, 20      'This task is important
TaskPriority hmiTask, 2      'This task is less important
Run myTask, hmiTask
Pause TaskStatus(hmiTask) = _tskTerminated
'Remaining code as the previous example
```

### Contact us

For more information please contact your
local ABB representative or one of the following:

**new.abb.com/motion**
**new.abb.com/drives**
**new.abb.com/drivespartners**
**new.abb.com/PLC**