AVIX Real Time Operating System

Getting Started with AVIX
A Tutorial

Product Version 5.0

AVIX-RT

AVIX
Advanced
Virtual
Integrated
eXecutive

AVIX-RT
Maïsveld 84
5236 VC 's-Hertogenbosch
The Netherlands

phone +31(0)6 15 28 51 77
e-mail info@avix-rt.com
www www.avix-rt.com

Table of contents

1 Introduction

AVIX comes with a tutorial application demonstrating many of the most important mechanisms offered. This tutorial offers a good starting point to become acquainted with AVIX. The tutorial application is optionally installed by the AVIX setup utility.

2 Tutorial Introduction

The tutorial illustrates the basic mechanisms of AVIX.

When installing the tutorial, in the root directory selected from the install utility, directory `_avix_tutorial` is created. All tutorial files are present in this directory. Start a tutorial by reading the content of file `README_tutorial.txt` which contains specific information for the port you are using.

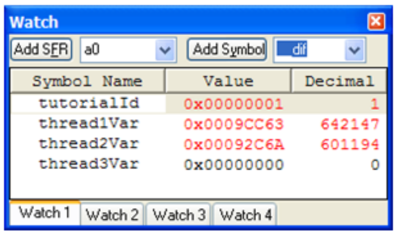
It is strongly advised to install the AVIX RTOS Viewer plug-in for the development environment you are using before running the tutorial application. This plug-in provides you with very valuable information about the AVIX based application you are developing.

2.1 What do the Tutorial Applications show?

The tutorial applications are all based on using four variables which are shown in the development environment watch window. As installed, the project files are set up such that these variables are already present in the watch window and you do not need to add them yourself.

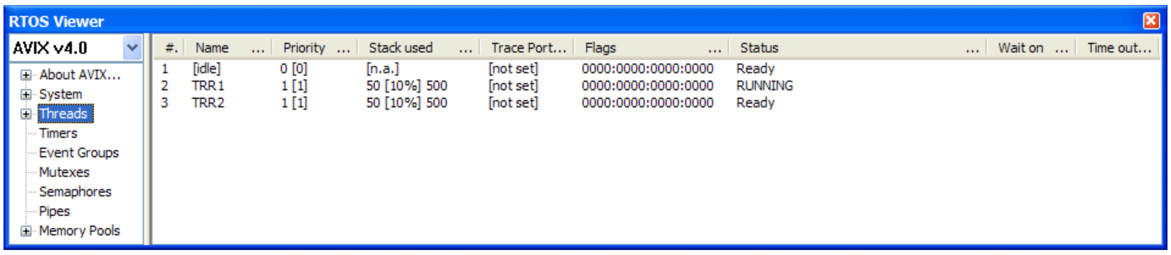
An AVIX based application is using multiple threads. Each tutorial application uses multiple threads where every thread manipulates one of these variables. The names of these variables are `thread1Var`, `thread2Var` and `thread3Var` and based on the specific tutorial application these variables are manipulated by the tutorial threads.

Besides these three variables there is another variable named `tutorialId`, also present in the watch window. When running one of the tutorial applications this variable holds an integer value identifying the currently running tutorial. In the figure to the right a sample of the watch window as present in the Microchip MPLAB8x environment is shown. The value of `tutorialId` is 1, meaning this screenshot belongs to tutorial1. Tutorial1 contains two threads manipulating variable `thread1Var` and variable `thread2Var` respectively. The third variable `thread3Var` is not manipulated by tutorial1
.



Symbol Name	Value	Decimal
tutorialId	0x00000001	1
thread1Var	0x0009CC63	642147
thread2Var	0x00092C6A	601194
thread3Var	0x00000000	0

As said before, it is strongly suggested to also install the AVIX RTOS Viewer plug-in. This provides invaluable information about an AVIX based application. The figure below shows this viewer as it looks in the Microchip MPLAB8x development environment with the thread window open. The content shown here is that of tutorial1 and the status of the two threads used in this tutorial is shown.



#.	Name ...	Priority ...	Stack used ...	Trace Port...	Flags ...	Status ...	Wait on ...	Time out...
1	[idle]	0 [0]	[n.a.]	[not set]	0000:0000:0000:0000	Ready		
2	TRR1	1 [1]	50 [10%] 500	[not set]	0000:0000:0000:0000	RUNNING		
3	TRR2	1 [1]	50 [10%] 500	[not set]	0000:0000:0000:0000	Ready		

2.2 What source files are there and how are they used?

The basic tutorial application contains the following source files:

_tutorialMain.c:
An AVIX based application starts by executing function `avixMain` which serves the same purpose as function main in a regular application. This function is contained in _tutorialMain.c. From this function the desired individual tutorial is activated. Below the content of function `avixMain` is shown:

```
 1 // Main function for AVIX based project. From here the specific
 2 // selected tutorial function is called.
 3 //
 4 void avixMain(void)
 5 {
 6    // Setup the system hardware (clock) and install an
 7    // AVIX central error handler. Note that
 8    //
 9    systemSetup();
10    avixError_SetHandler(myErrorFunc);
11
12    // Select the desired tutorial by setting the number of the desired tutorial
13    // in variable tutorialId
14    //
15    tutorialId = 1;
16
17    if (tutorialId ==  1) {t01RoundRobinScheduling(); }
18    if (tutorialId ==  2) {t02RoundRobinScheduling(); }
19    if (tutorialId ==  3) {t03Preemption();           }
20    if (tutorialId ==  4) {t04Timer();                }
21    if (tutorialId ==  5) {t05EventFlags();           }
22    if (tutorialId ==  6) {t06EventFlags();           }
23    if (tutorialId ==  7) {t07Messages();             }
24    if (tutorialId ==  8) {t08Pipes();                }
25    if (tutorialId ==  9) {t09Exchange();             }
26 }
```

Each individual tutorial is contained in a separate function in its own source file. The call to these functions is present on line 17 to 25 in the above code sample. A specific tutorial is activated by setting its number (1..9) in variable `tutorialId`. In the above code sample function `t01RoundRobinScheduling` is selected.

These functions contain the initialization code for the specific tutorial like creating threads and other AVIX kernel objects. When the tutorial specific initialization function returns, `avixMain` returns, which is the moment AVIX takes control and starts to schedule the threads that are created for the specific tutorial. The different tutorials are present in the following source files:

- **tutorial01_RoundRobin.c**: Round robin tutorial with individual thread functions.
- **tutorial02_RoundRobin.c**: Round robin tutorial with shared thread functions.
- **tutorial03_Preemption.c**: Preemption tutorial, higher priority thread preempting round robin threads.
- **tutorial04_Timer.c**: Timer tutorial, two threads waiting for a timer of 1ms and 2ms respectively.
- **tutorial05_EventFlags.c**: Event flag tutorial where threads set the event flag.
- **tutorial06_EventFlags.c**: Event flag tutorial where the event flag is set directly by a timer.
- **tutorial07_Messages.c**: Two threads sending data to third thread using messages.
- **tutorial08_Pipes.c**: Two threads sending data to a third thread using a pipe.
- **Tutorial09_Exchange.c**: Two threads sending data to a third thread using two Exchange objects.
- **systemSetup.c**: Controller specific initialization code called from `avixMain`.

2.3 Tutorial 1: Round Robin Scheduling, separate functions

What is demonstrated?

This tutorial illustrates the most basic scheduling principle offered by AVIX, Round Robin scheduling. Multiple threads having the same priority are scheduled in a Round Robin fashion. Each thread is active for a specific period, after which it is preempted and the next thread at that same priority becomes active. Although a thread (as any thread) looks like it is a completely autonomous program, still AVIX takes care all ‘programs’ (threads) are activated.

Description

This tutorial contains two threads at priority 1. Each contains an endless loop and continuously increments a variable. The code of one thread is shown in the code sample below.

```
// Function that will run as a thread under control of AVIX. This function
// updates t01_counterThread1 as can be observed in the watch window.
//
TAVIX_THREAD_REGULAR t01_thread1(void* p)
{
   while(1)
   {
      thread1Var++;
   }
}
```

Each of the two variables thread1Var and thread2Var will be updated. This update will alternate since the threads are alternating. Since the threads receive the same period, the value of the two variables will be the same.

Suggestions to learn more

Change priority of one thread: When you change the priority of one thread to a value of 2, you will see that the other thread will never be activated. AVIX will run the thread with the highest priority that is able to run. Since the threads in this tutorial never wait for any event, the thread with the highest priority will ‘monopolize’ the processor. This illustrates the principle that an RTOS based application works in an event driven fashion. Normally threads will not look like the threads in this tutorial but will wait for events. As will be shown further in this document, there are all kinds of events a thread can wait for in order to be activated when the event occurs.

Voluntarily give up the processor: A Round Robin thread will run for a specific period. Using function avixThread_Relinquish however a thread indicates it does not want to run for the entire period but allows the next thread in the Round Robin queue to run. Adding a call to avixThread_Relinquish to both threads after they updated their variable will alternate activation of the threads where each activation will only execute a single increment of the variable. So instead of thread1Var being incremented a number of times, followed by thread2Var being incremented a number of times, both will increment once. Doing so in this tutorial will introduce a lot of overhead since the threads are rescheduled taking something like 100 cycles and next do some useful work (increment the variable) for only a few cycles.

2.4 Tutorial 2: Round Robin Scheduling, shared function

What is demonstrated?

This tutorial illustrates the most basic scheduling principle offered by AVIX, Round Robin scheduling. Multiple threads having the same priority are scheduled in a Round Robin fashion. Each thread is active for a specific period, after which it is preempted and the next thread at that same priority becomes active. Although a thread (as any thread) looks like it is a completely autonomous program, still AVIX takes care all ‘programs’ (threads) are activated.

Description

This tutorial contains two threads at priority 1. Each contains an endless loop and continuously increments a variable. The difference with Tutorial1 is that in Tutorial2, the threads share the same code which is shown below. When creating a thread using avixThread_Create, a parameter can be passed which AVIX will pass as a parameter to the thread function when the thread is activated for the first time. Based on the value of this parameter the thread function can be coded to behave different.

This tutorial illustrates the difference between static and dynamic thread properties. The static part, the code, is the same for both threads. The difference is made in the dynamic part. Each thread has its own stack where the status of the individual thread is maintained.

Each of the two variables thread1Var and thread2Var will be updated. This update will alternate since the threads are alternating. Since the threads receive the same period, the value of the two variables will be the same.

In this tutorial the value of the two counter variables will be not as equal as with Tutorial1. Reason is that the code executed by both threads is slightly different. One thread will branch while the other doesn’t. This slight difference will result in slightly different counter values.

```
// Create two threads using the same function. The (void*)0 and (void*)1 parameter
// is passed to the thread function.
//
avixThread_Create("TRR1", t02_thread1, (void*)0, 1, 500, AVIX_THREAD_READY);
avixThread_Create("TRR2", t02_thread1, (void*)1, 1, 500, AVIX_THREAD_READY);



// Function that will run as a thread under control of AVIX.
// Parameter p is the (void*)0 or (void*)1 value used in avixThread_Create.
//
TAVIX_THREAD_REGULAR t02_thread1(void* p)
{
   while(1)
   {
      // Test whether I am thread 1 or thread 2. Based on which thread
      // I am, a different counter is updated.
      //
      if ( ((int)p) == 0)
      {
         thread1Var++;
      }
      else
      {
         thread2Var++;
      }
   }
}
```

2.5 Tutorial 3: Preemption

What is demonstrated?

The most important scheduling mechanism offered by AVIX is preemptive scheduling. When an event occurs a high priority thread is waiting for, AVIX will stop (preempt) the currently active thread in favor of the thread that received the event.

Description

This tutorial contains three threads. Two threads are running Round Robin with priority 1, just like in Tutorial1 and Tutorial2. A third thread with a higher priority (2) is sleeping. Sleeping is the most basic way of waiting for a time to expire. When a thread is sleeping (avixThread_Sleep), effectively it asks AVIX not to be activated until the specified time has elapsed. When the desired time elapses, this is an event for the thread that executed the sleep. Since the sleeping thread has a higher priority than the other two threads, AVIX will activate that thread the moment the time has elapsed. Doing so, this thread ‘preempts’ the currently running thread.

The two Round Robin threads will continuously update thread1Var and thread2Var. When the high priority thread starts running it will update thread3Var. Since this thread is sleeping for one millisecond, the value of thread3Var will be updated only once per millisecond.

```
// Function that will run as a thread under control of AVIX. This function
// updates thread3Var as can be observed in the watch window.
//
TAVIX_THREAD_REGULAR t03_thread3(void* p)
{
   while(1)
   {
      // This thread will sleep for one millisecond after which it wakes up.
      // Since the priority is higher than that of the other two threads,
      // this thread will preempt the thread currently active and run after
      // which it will sleep again.
      //
      avixThread_Sleep(AVIX_DELAY_US(100));

      thread3Var++;
   }
}
```

Suggestions to learn more

Timing accuracy: Thread3 sleeps for 1ms. Running the application until the value of thread3Var equals 1000, with the stopwatch functionality of your development environment you can see the application has been running for approximately 1.1 second. Effectively the sleep appears to be 1.1ms. This is caused by the resolution of the AVIX software timing mechanism. This resolution is equal to the system tick period. The system tick period is configurable and by default this is configured to be 100µs. When requesting a time, AVIX guarantees the requested time as a minimum. Requesting 1ms will effectively provide a delay between 1.0ms and 1.1 ms because of this system tick period. For precise timing, use must be made of explicit *cyclic* timers which are used in Tutorial4. Still, the resolution will be equal to the system tick period. This will be true for every RTOS, where AVIX has the added advantage the system tick period can be configured where most competing products use a fixed value of 1ms.

Change sleep period: Change the time thread3 will sleep to 100µs by using AVIX_DELAY_US(100) as the parameter value for avixThread_Sleep. Because of the above described ‘Timing Accuracy’ issue, the effective time will be 200 µs. You can verify this by letting the application run until the value of thread3Var equals 5000 and using the development environment stopwatch functionality you will see the application has run for 1 second then.

2.6 Tutorial 4: Timers

What is demonstrated?

For timing purposes AVIX offers timers. Timers can be set to a specific period and started. When the timer expires, this is an event a thread can wait for. Timers can be either 'single shot' or 'cyclic'. A 'cyclic' timer, once expired, will autonomously restart. Such a timer can be considered to expire over and over again without explicit restarting being required. Please read section 'Suggestions to learn more' of Tutorial3 to learn about the accuracy of timers.

Description

This tutorial contains two threads, thread1 waits for a cyclic timer with a period of 1ms and when running increments the value of thread1Var. Thread2 waits for a cyclic timer with a period of 2ms and when running increments the value of thread2Var. Below the code of thread1 is shown.

Because the timer of thread1 runs exactly twice as fast as that of thread2, the value of thread1Var will be exactly twice the value of that of thread2Var. As can be verified with the development environment stopwatch functionality, the effect described in section 'Suggestions to learn more' of Tutorial3 does not occur. Reason is the timers are cyclic timers. A timer always expires on the occurrence of the system tick and since the timer is restarted automatically, AVIX 'knows' the restart happens at the beginning of a system tick period resulting in exact timing.

```
// Function that will run as a thread under control of AVIX. This function
// updates thread1Var as can be observed in the watch window.
//
TAVIX_THREAD_REGULAR t04_thread1(void* p)
{
   tavixTimerId timerId;      // id the timer is represented by

   // Create a timer with a cyclic period of 1 ms.
   //
   timerId = avixTimer_Create("TMR1");
   avixTimer_Set(timerId, AVIX_DELAY_MS(1), AVIX_TIMER_CYCLIC);
   avixTimer_Start(timerId);

   while(1)
   {
      // Wait for the timer and update the counter. Since my period is 1 ms
      // and the other threads period 2 ms, my counter will be double that
      // of the other thread.
      //
      avixTimer_Wait(timerId);

      thread1Var++;
   }
}
```

Suggestions to learn more

Change timers to single shot: Change the timers to single shot type by using AVIX_TIMER_SINGLE_SHOT as the last parameter to avixTimer_Set. The timer must be explicitly restarted now in the thread body by adding a call to avixTimer_Start after the timer has expired. Now you will see the value of thread2Var will not be twice that of thread1Var. Reason is again that thread1 will wait for a time of 1.1ms and thread will wait for a time of 2.1ms. The value of thread1Var will be equal to the value of thread2Var multiplied by 2.1/1.1.

2.7 Tutorial 5: Event Groups, Explicit Usage

What is demonstrated

Event Groups are a mechanism for threads to synchronize and/or communicate binary information. Event Groups contain 16 binary flags. These flags can be set by a thread. Threads can also wait for one or more flags to be set. When waiting for multiple flags, the waiting thread can decide to wait until all desired flags are set or only one in a specific group of flags. Event Groups are kernel objects created through avixEventGroup_Create. Besides this, every thread has an implicit Event Group that is created when the thread is created.

Description

Starting point of this tutorial are the two threads of Tutorial4, one waiting for a 1ms timer and the other waiting for a 2ms timer. These threads still increment thread1Var and thread2Var respectively. This tutorial contains a third thread. When thread1 is active, it sets flag 0 in the implicit Event Group of thread3. When thread2 is active it sets flag 1 in the implicit Event Group of thread3. Thread3 waits for either flag to be set and when this is the case it runs and increments the value of thread3Var. As a result, thread3 will run when either thread1 or thread2 will run. As a result, the value of thread3Var will be the sum of thread1Var and thread2Var.

Besides usage of Event Groups, this tutorial demonstrates another important aspect of AVIX. To set flags in the Event Group of thread3, thread1 and thread2 need the id of thread3. This is obtained by giving thread3 a name when creating it. Using this name, thread1 and thread2 can execute an avixThread_Get function which returns the id of thread3.

An alternative would be to store the id of thread3 in a global variable. The advantage of using avixThread_Get however is that in case thread3 is not yet created when executing this function, the calling thread will block until thread3 does exist. So thread1 and thread2 are guaranteed to use a valid thread id. When using a global variable, the risk is thread1 or thread2 will use the content of this variable before thread3 exists which leads to an error.

The code fragment below shows the part of thread3 waiting for the desired flags and the subsequent processing.

```
// Every thread has got an implicit event group. Wait for either flag0 or
// flag1 to be set by the other threads.
//
flags = avixEventGroup_Wait
   (
      avixThread_GetIdCurrent().asEventId, // Convert thread id to event group id
      AVIX_EF(0) | AVIX_EF(1),             // Specify the flags to wait for.
      AVIX_EVENT_GROUP_ANY,                // Wake up when either flag is set.
      AVIX_EF_NONE,                        // Do no reset flags when starting to wait.
      AVIX_EF(0) | AVIX_EF(1)              // When wait is done, clear both flags.
   );

// The above function returns when either one flag or both flags are set.
// The actual flags set are returned in the functions result so now
// a test is done and for each of the flags set, the counter is updated.
// As a result, the count value of this thread will be the sum of that
// of the other threads.
//
if (AVIX_EF_IN(AVIX_EF(0), flags))   // Test if flag 0 is present in the result
{
   thread3Var++;
}
if (AVIX_EF_IN(AVIX_EF(1), flags))   // Test if flag 1 is present in the result
{
   thread3Var++;
}
```

Suggestions to learn more

Change wait from any flag to all flags set: The wait on line 4 of the above code fragment specifies any flag. So thread3 wakes up when either flag0, flag1 or both are set. When changing this to a wait for all flags (AVIX_EVENT_GROUP_ALL), thread3 will only wake up when both flags are set.

As a result, thread3 will only wake up with the rate of thread2, the thread waiting for 2ms. Although thread1 does set flag 0 every millisecond, thread3 will only react when also flag1 is set. So effectively one of every two set operations from thread1 will have no effect.

In this case the value of thread3Var will no longer be the sum of thread1Var and thread2Var but will be equal to thread2Var instead.

Make thread1 and thread2 free running: First, make sure the code is original again, so change the wait operation in thread3 back to any (AVIX_EVENT_GROUP_ALL). Next remove (by commenting out) the wait for timer operations in both thread1 and thread2. Now when the program is executed, still thread3 will wake on every activation of either thread1 or thread2. Still the value of thread3Var is the sum of the value of thread1Var and thread2Var with the difference that everything goes much faster now since thread1 and thread2 are free running.

See the effect of removing preemption: An interesting experiment now is to lower the priority of thread3 to 1, the same as that of thread1 and thread2. The effect is that when either thread1 or thread2 set a flag, thread3 does not preempt and thread1 or thread2 are allowed to continue for the duration of the Round Robin time slice.

So when either of the flags is set, all that happens is that thread3 is no longer waiting but will be placed at the end of the queue for priority 1. Only when both thread1 and thread2 have finished their Round Robin time slice, thread3 will execute. Thread3 will update thread3Var and start waiting again. In the Round Robin time slice they are allowed to consume, thread1 and thread2 constantly set the flag but since thread3 does not preempt, these threads are allowed to continue. As a result thread3 will only 'see' one flag set out of many and the value of thread3Var will be much lower than that of thread1Var and thread2Var.

2.8 Tutorial 6: Event Groups, Implicit Usage

What is demonstrated

Event Groups are a mechanism for threads to synchronize and/or communicate binary information. Event Groups contain 16 binary flags. These flags can be set by a thread. Threads can also wait for one or more flags to be set. When waiting for multiple flags, the waiting thread can decide to wait until all desired flags are set or only one in a specific group of flags. Event Groups are kernel objects created through avixEventGroup_Create. Besides this, every thread has an implicit Event Group that is created when the thread is created.

Besides Event Flags being set by an explicit operation of a thread, it is also possible to ‘connect’ an Event Group to a timer. As a result, the specified Event Flags are changed when the timer expires without intervention of a thread. Changing the flags on timer expiration is entirely taken care of by AVIX.

This is a very useful mechanism allowing a thread to wait for multiple events. By ‘connecting’ an Event Flag to a timer a thread can wait for this Event Flag and other Event Flags which are explicitly set in a single wait. AVIX also allows Event Flags to be ‘connected’ to message queues so a thread can wait in a single wait call for one or more timers, messages and other flags.

Description

This tutorial is equal to tutorial5 with the difference that thread1 and thread2 do not contain a function call to set an Event Flag. Instead of this, thread1 ‘connects’ the Event Group of thread3 to its timer to automatically set flag 0 when the timer expires. Likewise, thread2 ‘connects’ the Event Group of thread3 to its time to automatically set flag1 when the timer expires. For the rest the behavior equals that of Tutorial5.

The code sample below shows the code in thread1 where an Event Group is ‘connected’ to the timer.

```
// The id of the thread is connected to the timer so when the timer
// expires it will automatically set the flag and this is no longer
// required by the thread code to be done.
//
//
avixTimer_ConnectEventGroupThread
   (
      timerId,                 // When timer 'timerId' expires
      flagThread,              // in the event group of thread 'flagThread'
      AVIX_EVENT_GROUP_SET,    // set is
      AVIX_EF(0)               // flag 0 without the thread needing to do this
   );
```

2.9 Tutorial 7: Message Mechanism

What is demonstrated

This tutorial demonstrates the message passing mechanism. Message passing is a mechanism to send data from one thread to another. A message is a block of memory allocated from a pool that can be filled by the sending thread with data of any structure and subsequently send to another thread. The receiving thread waits for the message, reads the data and processes it. Next, the receiving thread discards the message memory block to be reused. Besides the data section of a message, every message has a fixed type field. This is a numeric field containing a user defined value to identify the message so the receiver knows what the meaning of the message is.

Description

Starting point of this tutorial are two threads, one waiting for a 1ms timer and the other waiting for a 2ms timer. These threads increment thread1Var and thread2Var respectively. Besides updating their count variables, both threads allocate a message, fill the message with their count value and send the message to a third thread (thread3). This thread waits for the messages and calculates the average of the count values and writes this average value to thread3Var.

The code sample below shows how a sending thread allocates a message, fills it with its counter value and sends it to the message queue of the receiving thread.

```
// Allocate a message, fill it with the counter value and send to the third thread.
//
msg = avixMsg_Allocate(2);            // Allocate a message and give it id 1 so
                                      // the receiver knows the origin.

avixMsg_PutLong(msg, thread2Var);     // Set the counter value in the message.

avixMsgQThread_Send(msg, msgThread); // Send the message
```

The receiving thread waits until it has at least once received a message from both sending threads and next calculates the average of these values to be placed in thread3Var. The code fragment of the receiving thread is shown in the code sample below.

```
// I know to expect a type 1 and a type 2 message so now check which one this is.
//
if (avixMsg_GetType(msg) == 1)
{
   msg1received = 1;                // So the thread1 data is received, set flag
   count1 = avixMsg_GetLong(msg);   // and read the value from the message.
}

if (avixMsg_GetType(msg) == 2)
{
   msg2received = 1;                // So the thread2 data is received, set flag
   count2 = avixMsg_GetLong(msg);   // and read the value from the message.
}

// Free the message so it can be reused again
//
avixMsg_Free(msg);

// When data of both threads is received, the average is calculated and set in
// the variable so it can be watched.
//
if (msg1received && msg2received)
{
   thread3Var = (count1 + count2) / 2;
}
```

2.10 Tutorial 8: Pipe Mechanism

What is demonstrated

This tutorial demonstrates the pipe mechanism. Pipes are First-In First-Out queues of data that can be used to exchange data between threads and Interrupt Service Routines and threads. A pipe contains blocks of data. The size of a block is user defined so pipes can be used not only to hold char or int data but also data of a user defined type. When a pipe is full, a thread writing to the pipe is blocked. When the pipe is empty, a thread reading from the pipe is blocked.

Description

Starting point of this tutorial are two threads, one waiting for a 1ms timer and the other waiting for a 2ms timer. These threads increment thread1Var and thread2Var respectively. Besides updating their count variables, both threads write their count value to a pipe. The other end of the pipe is read by a third thread (thread3) which calculates the average of the count values and writes this average value to thread3Var.

The pipe is created by thread3 since this is a thread ‘servicing’ the other two threads and thus the pipe can be considered an ingoing interface for this thread.

Just like in the message tutorial, the receiving thread must know it has received data from both sending threads. To accomplish this, the type of the block of data the pipe holds is a struct holding besides the count value an integer value with an id of the sending thread.

The code sample below shows how the pipe is created using the sizeof of the user defined struct as the blocksize. The pipe is given a name so the sending threads can get access to the pipe using function avixPipe_Get which returns the id of the pipe based on the name it is given during construction.

```
// Create the pipe used by the other two threads to send me data. Through the
// name given here, the other two threads can obtain the id of the pipe.
// The blocksize depends on the struct defined for the data.
//
pipe = avixPipe_Create("PIP", 5, sizeof(tPipeData), NULL, NULL);
```

The code sample below shows how a sending thread writes a data block to the pipe by first filling the data block both with the id and the count value and subsequently writing the data to the pipe.

Note that the third parameter to the write function is 1, denoting that a single block is written. The pipe ‘knows’ the number of bytes involved since this is specified when creating the pipe.

```
// Put id + data in a struct that is written to the pipe.
//
data.id = 2;                // Let the receive know the data comes from thread 2
data.value = thread2Var;    // The actual data.

// Pipes are block oriented and know the size of the data blocks they hold
// since one block is written, the third parameter is 1.
//
avixPipe_Write(pipe, (unsigned char*)&data, 1);
```

The receiving thread waits for the pipe. The data read from the pipe is placed in a struct of the correct type which is a local variable in the thread function.

The code sample below shows the receiving process:

```
// Wait for a pipe to hold a single block.
//
avixPipe_Read(pipe, (unsigned char*)&data, 1);


// I know to expect a type 1 and a type 2 data block so now check which one this is.
//
if (data.id == 1)
{
   data1received = 1;           // So the thread1 data is received, set flag
   count1 = data.value;         // and get the data from the struct read from the pipe.
}

if (data.id == 2)
{
   data2received = 1;           // So the thread2 data is received, set flag
   count2 = data.value;         // and get the data from the struct read from the pipe.
}


// When data of both threads is received, the average is calculated and set in
// the variable so it can be watched.
//
if (data1received && data2received)
{
   thread3Var = (count1 + count2) / 2;
}
```

2.11 Tutorial 9: Exchange Mechanism

What is demonstrated

This tutorial has the same functionality as tutorial 7. In tutorial 7, two threads send a message with a counter value to a third thread which calculates the average value of the received counter values and sets this in a third variable to watch in the watch window.

This tutorial 9 does not send messages directly but uses exchange objects instead. So the two threads producing count values do not send a message but just write the count value to an Exchange object. The receiving thread is connected to both Exchange objects so when a thread writes to its Exchange, thread 3 will receive a message.

Although this may not seem to be an important difference, it is. Using the Exchange objects, the sending threads do not know which thread is listening to the Exchange. They just write data and are done. This decouples the senders code wise from the receiver. Furthermore, the senders do not even know how many threads are 'listening' to the Exchange. So if the information is received by one or more threads makes no difference for the code of the senders.

Using messages directly like in tutorial 7 would imply the senders to be changed since in that case they had to explicitly send the messages to multiple receivers.

So adding additional receivers means adding code but not changing existing code.

Description

Starting point of this tutorial are two threads, one waiting for a 1ms timer and the other waiting for a 2ms timer. These threads increment thread1Var and thread2Var respectively. Besides updating their count variables, both threads write their count value to an Exchange object. Thread3 is connected to both Exchange objects with a message connection. This means that when a new value is written to the Exchange object, a message is generated and send to the connected thread3.

This thread waits for the messages and calculates the average of the count values and writes this average value to thread3Var. The code sample below shows how a thread writes its counter value to the Exchange object.

```
//-----------------------------------------------------------------------------------
// Write the counter value to the Exchange. The cast is to prevent the warning
// resulting from the variable being volatile.
//-----------------------------------------------------------------------------------
avixExch_Write(exchange, (long*)&thread1Var);
```

The receiving thread connects to the two Exchange objects and next waits until it has at least once received a message from both connections and next calculates the average of these values to be placed in thread3Var. The code fragment of the receiving thread is shown in the code sample below. The code sample below shows how the receiving thread connects to the Exchange objects.

```
//-----------------------------------------------------------------------------------
// Obtain the id of the Exchanges I will connect to and create a message connection.
// Note that here the decission is made that when new data is written to the Exchange,
// this will result in a message being send to this thread!
//-----------------------------------------------------------------------------------
exchange = avixExch_Get("EXC1");
avixExch_ConnectMsgQThread(exchange, avixThread_GetIdCurrent(), 1);

exchange = avixExch_Get("EXC2");
avixExch_ConnectMsgQThread(exchange, avixThread_GetIdCurrent(), 2);
```