The Multi-Threading Tutorial (Part I)

- The Multi-Threading Tutorial (Part I)

Part I of the Multi-Threading Tutorial discusses fundamental principles for multi-threading from the "program logic" point of view. It is intended to give you the knowledge to effectively use multiple threads in your Xbase^{++} programs and to take full advantage of multi-threading. The source code discussed in Part I of this tutorial is available for download from Alaska Software's home page (MTTUTOR1.ZIP).

Introduction

As you know, multi-threading is a key feature of 32-bit operating systems and every developer who can master this technology not only makes their life easier but has an enormous advantage. If you use multi-threading wisely, you can produce user-friendly applications which include features that will answer your (future) customer's question whether "to buy or not to buy" your software. If you have previously used multi-threading in other programming languages, try to forget how frustrating and complicated this process was. We will discuss typical problems associated with multi-threading and provides elegant solutions for multi-threading.

Multi-threading is known to be complicated in other programming languages, but with Xbase^{++} you have this feature at your finger tips. There are many things you don't need to care about - because Xbase^{++} does them for you. You can use multiple threads right out of the box without problems. However, you will need to be somewhat familiar with the concept of multi-threading to avoid improper use of program logic required when different parts of a program run in different threads at the same time.

The classical example of multi-threading is a program that lets a user enter data while a print job is being processed in the background at the same time. There is no problem if data entry and printing is implemented in two threads. But there are more exciting examples where multiple threads are useful, such as animation, tooltips, agents, watchdogs or incremental search in browsers, just to name a few. What would you think if an application reminds you five minutes in advance that you have an appointment with your boss at 4:15pm while you are concentrating on other tasks at your desk or computer. Of course, there are products specialized for this task, but wouldn't your customers appreciate this feature without having such a specialized product? If you know how to do it, you can add such a feature to your application in less than 2 hours using Xbase^{++}. The major task of this tutorial is to explain *program logic* required in multi-threading and to open the tremendous possibilities for solving common programming problems by using multiple threads.

Displaying animated bitmaps

This section discusses a variety of multi-threading issues using animation as an example. Animated graphic images have become popular because they attract the user's attention and provide for some kind of entertainment. This can be advantageous but can also have the opposite effect of distracting the user from their original task while using the software. Just as in real life, the rule "too many is too much" applies to animated images as well, and the technique should be used sparsely in an application. If used in the right places, however, animated images can be very informative for the user. So every developer should know how animated images are programmed and how this is achieved in the easiest way. Besides this, animation is a perfect topic to hilight different aspects of a multi-threaded program.

The fundamental technique

An animation consists of a series of single images each of which shows a distinct phase of the animation. We are using bitmaps for the animation and the first thing to start with is creating the bitmaps. An example of a series of three bitmaps is shown below.

Three phases of an animation

A black and white circle rotated twice by 30 degrees is not very exciting but is a sufficient example to demonstrate animation and multi-threading. A user gets the impression of a rotating circle when the three bitmaps are displayed one after the other at the same place and that is basically the whole story of animation: displaying different images (or phases) at the same or changing position. Once the series of bitmaps is available, we need the following two classes to bring the animation to life:

XbpBitmap	One XbpBitmap object is required to load one bitmap file and display its contents. We will create three of these objects for a three-phased animation and collect them in an array for easy access.
XbpStatic	An XbpStatic object is used as a kind of canvas where the bitmaps are drawn. It provides for the presentation space required by XbpBitmap objects when displaying the image.

Knowing these two classes we can put the pieces together and discuss the basic technique for an animation using a simple program:

```
01: PROCEDURE Main
02:    LOCAL oXbp, aBitmaps
03:
04:    Setcolor( "N/W" )
05:    CLS
06:
07:    oXbp     := XbpStatic():new( ,, {10,300}, {44,44} )
08:    oXbp:create()
09:
10:    aBitmaps := PrepareAnimation( { "Phase1.bmp", ;
11:                                    "Phase2.bmp", ;
12:                                    "Phase3.bmp"  } )
13:    DO WHILE .T.
14:       Animate( oXbp, aBitmaps )
15:       Sleep( 10 )
16:    ENDDO
17: RETURN
18:
19:
20: // Loads bitmap files
21: FUNCTION PrepareAnimation( aFiles )
22:    LOCAL i, imax := Len( aFiles )
23:    LOCAL aBitmaps:= Array( imax )
24:
25:    FOR i:=1 TO imax
26:       aBitmaps[i] := XbpBitmap():new():create()
27:       aBitmaps[i]:loadFile( aFiles[i] )
28:    NEXT
29: RETURN aBitmaps
30:
31:
32: // Displays a collection of bitmaps
33: PROCEDURE Animate( oXbp, aBitmaps )
34:    STATIC nCurrent := 0
35:    LOCAL oPS := oXbp:lockPS()
36:
37:    nCurrent ++
38:    IF nCurrent > Len( aBitmaps )
39:       nCurrent := 1
40:    ENDIF
41:
42:    aBitmaps[ nCurrent ]:draw( oPS, {1,1} )
43:    oXbp:unlockPS( oPS )
44: RETURN
```

The bitmap files participating in the animation are loaded into the program by XbpBitmap objects in a separate function *PrepareAnimation()*. The function receives an array of bitmap file names and creates for each file an XbpBitmap object in line #26 which in turn loads the bitmap file. The objects are collected in an array which is returned.

The animation is executed by calling procedure *Animate()* continuously within a DO WHILE loop (line #14). The procedure receives the XbpStatic object as the place *where* to draw the bitmaps, and the array containing XbpBitmap objects which know *how* to draw a bitmap. A bitmap becomes visible on the screen in line #42 where the *:draw()* method of an XbpBitmap object is called. Once a bitmap is drawn, the program pauses for 0.1 seconds in line #15 (the unit for the Sleep() function is one hundredth of a second).

The key for the animation is the variable *nCurrent*. It is declared as STATIC (line #34) and retains its value when procedure Animate() returns. The only thing necessary for displaying the next bitmap of the animation is, therefore, to increment the STATIC variable and reset it to One if its value exceeds the number of available bitmaps (line #37 through #40). As a result, each call to procedure Animate() displays another bitmap.

This program demonstrates the basic techniques required for programming an animation: one XbpBitmap object is created for each phase of the animation. Each XbpBitmap object loads a single bitmap file and draws the image in another Xbase Part. This Xbase Part must know the method *:lockPS()* which returns a presentation space required by an XbpBitmap object for drawing its bitmap. The program uses an XbpStatic object as a canvas but it could be any Xbase Part derived from *XbpWindow()* (*:lockPS()* is a method in the *XbpWindow()* class that is inherited by any object derived from *YbpWindow(), such as XbpStatic()*). You can draw an animation in a pushbutton, for example, when you use an XbpPushbutton object instead of an XbpStatic object. If you change XbpStatic() in line #7 to XbpPushbutton(), you see the animation within a pushbutton. So, it lies within your imagination how to use this technique in your applications.

Identifying program code for a thread

The example program cannot be used for anything but loading and displaying bitmaps. As a matter of fact, the program cannot be stopped unless you press Alt+C. So, what is its purpose in the multi-threading area? The answer lies in the question: Why is the program code separated into *PrepareAnimation()* and *Animate()*? The whole animation could have been programmed in *Main()*. To find the answer, take the *program logic* point of view and think what makes *PrepareAnimation()* logically different from *Animate()*? The difference is that *PrepareAnimation()* is called once while *Animate()* is called multiple times, and this is how the example program is structured: The part which needs to be called once is separated entirely from the part that must be called multiple times. This again leads to a key question you have to answer when using multi-threading: **how often is a procedure or function called?** Answering this "key question" will help to structure your multi-threaded programs.

To make the example a bit more useful we will allow for user input while bitmaps are being displayed. The easiest way to accomplish this is @..SAY..GET followed by the READ command, which is as good as any other approach for obtaining user input in this discussion.

We add a new level of complexity to the program and it consists now of two completely different things:

```
User input                      Animation

@ 10, 10 SAY "X" GET varX       DO WHILE .T.
@ 12, 10 SAY "Y" GET varY          Animate( oXbp, aBitmaps )
                                   Sleep( 10 )
READ                            ENDDO
```

This is an ideal situation: something requires user input and something else does not. In fact, an animation must run independently of user interaction and this makes it a perfect candidate for a separate thread. The main thread allows for data entry in the program while a second thread is busy with displaying bitmaps. The second thread, however, will execute only that part of the animation which must be executed repeatedly.

Starting a thread

Once the program code that can be run in a separate thread is identified, we can encapsulate it in a procedure and let a Thread object handle the program execution. A Thread object represents an additional thread, or execution path, so that two procedures can be executed at the same time. This requires only few modifications in our example program:

```
01: // User enters data in Main
02: PROCEDURE Main
03:    LOCAL oXbp, aBitmaps, oThread
04:    LOCAL cFirst := "Henry  ", cLast := "Miller "
05:
06:    Setcolor( "N/W,W+/B" )
07:    CLS
08:
09:    oXbp     := XbpStatic():new( ,, {10,300}, {44,44} )
10:    oXbp:create()
11:
12:    aBitmaps := PrepareAnimation( { "Phase1.bmp", ;
13:                                    "Phase2.bmp", ;
14:                                    "Phase3.bmp"  } )
15:
16:    oThread := Thread():new()
17:    oThread:start( "ExecuteAnimation", oXbp, aBitmaps )
18:
19:    SET CURSOR ON
20:    @ 10, 10 SAY "Firstname:" GET cFirst
21:    @ 12, 10 SAY " Lastname:" GET cLast
22:    READ
23: RETURN
```

```
24:
25:
26: // This runs in a separate thread
27: PROCEDURE ExecuteAnimation( oXbp, aBitmaps )
28:    DO WHILE .T.
29:       Animate( oXbp, aBitmaps )
30:       Sleep( 10 )
31:    ENDDO
32: RETURN
```

The effect of this program is that a user can enter data while three bitmaps are continuously displayed in a round robin scheme. The code for loading and displaying the bitmaps is not listed here because it is the same as discussed in The fundamental technique. The important changes are:

1. The Main() procedure covers user interaction and allows for data entry using the READ command in line #22.
2. The DO WHILE loop is moved to a separate procedure (line #28 through line #31), so that it can be executed by the Thread object created in line #16.

The example program consists now of two threads and both execute program code performing two completely different tasks: data entry versus display of bitmaps. This situation is perfect for multi-threading since both tasks have nothing in common. The two threads use different memory variables and different code which is the best (or easiest) situation a programmer can have in multi-threading. The only task required is creating a Thread object and telling it what program code to execute. This is done by calling the *:start()* method (line #17) which receives as first parameter the name of the function/procedure to be executed in the new thread. All following parameters passed to *:start()* are just passed on to the called procedure.

Since the program flow is not obvious from the program code it must be emphasized that the DO WHILE loop is executed **at the same time** as the READ command. This is something the operating system takes care of and clearly reveals the nature of multi-threading. It also shows the superiority of the multi-threaded approach over a single-threaded solution. It is possible to display bitmaps every 0.1 seconds while READ is executed in a single-threaded application. It is impossible, however, to program this with less code. One would have to hook into the Get system using a customized Get reader or would need to modify the Get system accordingly. Both approaches result in an unnecessary programming overhead and would create a logical dependency between two tasks which don't have anything in common.

Stopping a thread

Although the example program is now capable of performing two tasks simultaneously (user input and animation) it has a major disadvantage: the thread displaying the animation cannot be stopped when the READ command is finished. This does not matter in the example program because the READ command is followed by the RETURN statement which ends

the entire program, including the second thread. But what if some other code would follow the READ command? The animation would continue to run and there is absolutely no way to stop the thread displaying bitmaps because of the DO WHILE .T. condition. This loop runs forever and we must find a way to stop the animation, or thread.

Stopping a thread is not as simple as starting it because a Thread object does not have a *:stop()* method, there is only a *:start()* method. A Thread object terminates its thread automatically if the code has run to completion in the thread. In other words, if a RETURN statement is executed in that part of a program which is invoked via the *:start()* method.

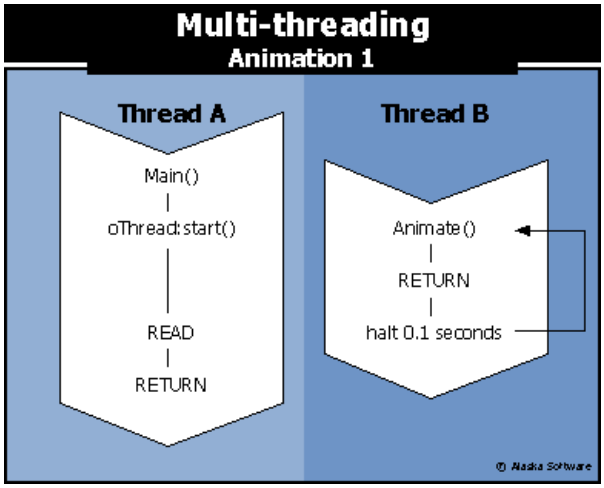
The RETURN statement is never reached in procedure ExecuteAnimation() of the example. The only possibility for this is to exit the DO WHILE .T. loop. This could be achieved by using a PUBLIC variable serving as logical condition for the loop. The variable would have to be PUBLIC because it must be visible in two threads. The first thread would set this variable to .T. before the second thread starts, and would set it to .F. in order to exit the loop. This in turn would cause the second thread to terminate.

At first thought, this is a feasible scenario but there is a more elegant solution which uses a special feature of the Thread object. Have a look at the modified example program below which uses special features of a Thread object:

```
01: // User enters data in Main
02: PROCEDURE Main
03:    LOCAL oXbp, aBitmaps, oThread
04:    LOCAL cFirst := "Henry  ", cLast := "Miller "
05:
06:    Setcolor( "N/W,W+/B" )
07:    CLS
08:
09:    oXbp     := XbpStatic():new( ,, {10,300}, {44,44} )
10:    oXbp:create()
11:
12:    aBitmaps := PrepareAnimation( { "Phase1.bmp", ;
13:                                    "Phase2.bmp", ;
14:                                    "Phase3.bmp"  } )
15:
16:    oThread := Thread():new()
17:    oThread:setInterval( 10 )
18:    oThread:start( "Animate", oXbp, aBitmaps )
19:
20:    SET CURSOR ON
21:    @ 10, 10 SAY "Firstname:" GET cFirst
22:    @ 12, 10 SAY " Lastname:" GET cLast
23:    READ
24:
25:    oThread:setInterval( NIL )
26:    oThread:synchronize( 0 )
```

```
27:
28:    WAIT "Thread has stopped"
29: RETURN
30:
31:
32: // Displays a collection of bitmaps
33: PROCEDURE Animate( oXbp, aBitmaps )
34:    STATIC nCurrent := 0
35:    LOCAL oPS := oXbp:lockPS()
36:
37:    nCurrent ++
38:    IF nCurrent > Len( aBitmaps )
39:       nCurrent := 1
40:    ENDIF
41:
42:    aBitmaps[ nCurrent ]:draw( oPS, {1,1} )
43:    oXbp:unlockPS( oPS )
44: RETURN
```

The solution is that a DO WHILE loop is not required at all when program code must be executed repeatedly in a thread. This is an important shift in program logic and becomes possible due to a Thread object's intelligence. The key to the program logic is line #17 where a time interval of 10 hundredths of a second (0.1 seconds) is set for the Thread object to restart the Animate() procedure. This procedure is executed in the new thread and called for the first time in line #18. Once the procedure is finished, it is restarted automatically after 0.1 seconds by the Thread object. This means that program code is executed repeatedly which runs parallel to the READ command:



From the *program logic* point of view it is important to understand that while the READ command is executed in thread A (line #23) the Animate() procedure is executed entirely in

thread B from line #33 down to line #44. The RETURN statement is **really** executed in thread B but this does not end the thread. Instead, thread B is just halted for 0.1 seconds due to the time interval set. It resumes with executing Animate() again when the interval has elapsed. The thread consumes no system resources while it pauses, it is -literally spoken- put to sleep for 0.1 seconds.

The *:setInterval()* method of the Thread object is the easiest way to achieve repeated execution of the same program code in a thread, once a thread is started. This again shows a major difference in program logic compared to single-threaded programs: Instead of using a DO WHILE loop for code repetition, a time interval is defined which causes the Thread object to execute program code again when the time interval has elapsed. Of course, we could use a DO WHILE loop as well, but there is one big advantage using the *:setInterval()* approach: the time interval can be voided and this is our chance to stop a thread easily:

```
25:      oThread:setInterval( NIL )
26:      oThread:synchronize( 0 )
```

These two lines allow you to effectively stop thread B from thread A because thread B does not execute the Animate() procedure again when the interval is set to NIL. The *:synchronize()* method accepts as a parameter a time-out value. Passing the value zero to this method means: there is no time-out condition. This causes thread A to wait forever until thread B has ended. This again is the only way to be sure that thread B is no longer running and only then may thread A resume with program execution.

Calling the *:synchronize()* method in line #26 makes sure that thread A waits until thread B has terminated. This is something you must be aware of when using multiple threads. If you want to stop a thread you have to assure in your program that the thread you want to stop has definitely ended. Otherwise you will have a good chance of getting inconsistent runtime errors in your multi-threaded programs. One time your application bombs, or ends unexpectedly, but when you restart it to find the error it just runs fine. Such an occasional runtime error is a worst case scenario in multi-threaded programs and you are well advised to avoid this kind of problem right from the beginning. You should keep in mind, therefore, that "stopping a thread" means **be sure that the thread has ended**.

Making it thread-safe

The example program is now in a stage where we can start an animation, retrieve user input while the animation is running, stop the thread and restart it if this is necessary. However, there is still one major design flaw that makes the example unsuitable for reuse. Just recall how the animation is displayed:

```
01: // Displays a collection of bitmaps
02: PROCEDURE Animate( oXbp, aBitmaps )
03:    STATIC nCurrent := 0
04:    LOCAL oPS := oXbp:lockPS()
05:
06:    nCurrent ++
07:    IF nCurrent > Len( aBitmaps )
08:       nCurrent := 1
09:    ENDIF
10:
11:    aBitmaps[ nCurrent ]:draw( oPS, {1,1} )
12:    oXbp:unlockPS( oPS )
13: RETURN
```

Since the Animate() procedure is entirely executed before it is called again, the usage of a STATIC variable for tracing the current bitmap of the animation is obvious at first sight. A STATIC variable retains its last value and we get the next bitmap to be displayed in a new execution cycle by simply incrementing the STATIC variable *nCurrent* in line #6. This is an absolutely correct implementation and a sound program logic as long as there is only one animation running.

But what happens if two animations are displayed using two threads? The STATIC variable *nCurrent* would be incremented alternating from two threads and this would spoil both animations since the sequence of bitmap display is not guaranteed in either case. So, this implementation is not thread-safe because procedure Animate() is **not re-entrant**. It cannot be called simultaneously from more than one thread.

This situation is something you will most probably run into if you are not familiar with multi-threading, and you must be aware of it! Remember the key question: **How often is a procedure or function called?** This question includes not only how often something is called in one thread, but also **How many threads execute the same code simultaneously?** If a program code runs perfectly in one thread, it does not mean that it will work simultaneously in many threads. Take a look back to the discussion in the The fundamental technique section where the STATIC variable *nCurrent* is said to be "the key" for an animation. In fact, using a STATIC variable is a great idea from the "animation" point of view, but when we look at the program logic from the "multi-threading" point of view, we have to come up with a better idea.

The solution for the problem is to program only functions etc. which are re-entrant. This means that the value of variables a function relies on must not be changed in different threads at the same time. STATIC variables are in most cases unsuitable for multi-threading and we have to change the implementation of the Animate() procedure. The easiest way to replace the variable *nCurrent* is the array *aBitmaps*.

The problem is solved when the array contains not only XbpBitmap objects but also the array index pointing to the current bitmap:

```
{ Array index, { XbpBitmap1, XbpBitmap2, XbpBitmap3 } }
```

Using an array of this structure, the Animate() procedure becomes thread-safe and looks as follows:

```
01: // Displays a collection of bitmaps
02: PROCEDURE Animate( oXbp, aBitmaps )
03:    LOCAL oPS := oXbp:lockPS()
04:
05:    aBitmaps[1] ++
06:    IF aBitmaps[1] > Len( aBitmaps[2] )
07:       aBitmaps[1] := 1
08:    ENDIF
09:
10:    aBitmaps[ 2, aBitmaps[1] ]:draw( oPS, {1,1} )
11:    oXbp:unlockPS( oPS )
12: RETURN
```

The result of this implementation is that all data required for an animation is stored in one array. Two animations, or threads, respectively, use two different arrays holding different data for each animation. This means that procedure Animate() can be called from different threads, but each thread uses its own array when Animate() is executed. The key for the program logic is now that each thread gets its own set of data because different arrays arrive in the parameter *aBitmaps* when the procedure is called simultaneously from multiple threads.

Building an Animation class

This section focuses on the various aspects for defining a user-defined Thread class that inherits the ability to manage a thread from the built-in Thread class. Inheriting from the Thread class is just as easy as from any other class. However, there are some implementational rules that must be followed for successfully using objects of a user-defined Thread class. The example of displaying animated bitmaps is discussed again, using an object-oriented approach.

Specifications

Before we begin to implement a class we have to find a name for it. A good one is *Animation* because this is a synonym for what objects of the class will do in general. The next step is to define what data an Animation object will *have*, or *know*, and what it will *do* with the data. You have a pretty good idea about this from the example program in the previous section and should try to define the specifications on your own. Each animation, for example, must maintain its own array index for selecting the current bitmap, each must have its own XbpBitmap objects and so on. Since different data is required in different threads it becomes obvious that we must tell a Thread object what data to use for the animation and store the corresponding values directly in the object. Now this is what instance variables are good for: holding the data required for processing an animation. Instance variables have a symbolic name used to access their value and it is a good idea to specify not only the required data but also the names of the corresponding instance variables.

Data is used in the methods of an object and the different stages of an animation must be implemented in different methods. The entire animation is split into a number of methods, which also have symbolic names. The following table shows both the name and description of member variables (data) and methods (functionality) for the Animation class.

Specification of an Animation class

MemberVar/Method	Description
Data (Instance variables) for an Animation object (what it knows)	
:aSource	Array holding names of bitmap files or their resource IDs if a resource file is used
:cDllName	Name of DLL if a resource file is linked to a DLL
:aBitmaps	Array holding XbpBitmap objects
:nTotal	Total number of bitmaps for the animation
:nCurrent	Array index pointing to the current bitmap
:aRect	Coordinates where to draw the bitmaps in an Xbase Part

MemberVar/Method	Description
Functionality (Methods) of an Animation object (what it does)	
:init()	Creates a new thread
:atStart()	Loads bitmap files when the thread starts
:execute()	Displays bitmap files while the thread is running
:atEnd()	Releases bitmaps before the thread ends
:stop()	Stops the animation
Inherited from the Thread class	
:start()	Starts the animation

Pre-requisites for program logic

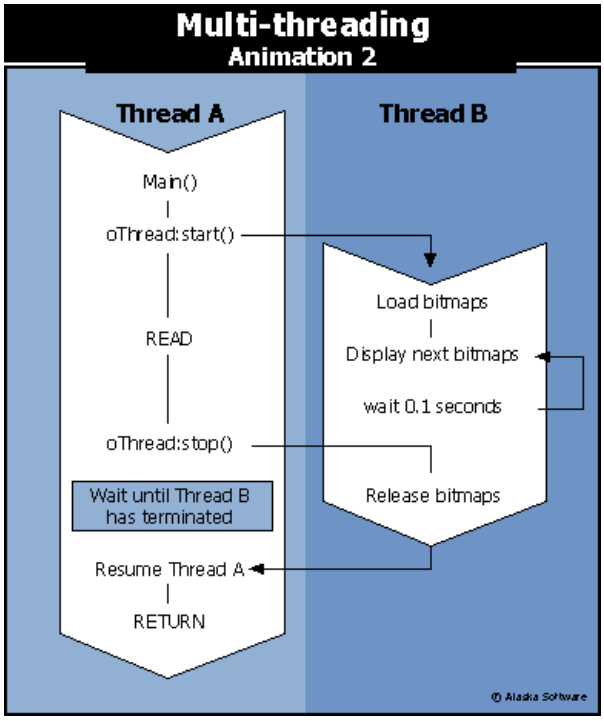
The specifications are detailed enough for declaring and implementing the Animation class. The key for this class to work properly is the built-in Thread class which enables objects of the Animation class to manage their own thread. As a matter of fact, each Animation object is a thread that knows how to display bitmaps. An Animation object uses data stored in member variables and processes them in its methods. For a better understanding of the implementation of the Animation class, let us first see how an object of this class is used:

```
01: // User enters data in Main
02: PROCEDURE Main
03:    LOCAL cFirst := "Henry  ", cLast := "Miller "
04:    LOCAL oXbp, oThread
05:
06:    Setcolor( "N/W,W+/B" )
07:    CLS
08:
09:    oXbp := XbpStatic():new(,, {10,300},{44,44} )
10:    oXbp:create()
11:
12:    oThread := Animation():new( { "Phase1.bmp", ;
13:                                  "Phase2.bmp", ;
14:                                  "Phase3.bmp"  } )
15:    oThread:start( , oXbp )
16:
17:    SET CURSOR ON
18:    @ 10, 10 SAY "Firstname:" GET cFirst
19:    @ 12, 10 SAY " Lastname:" GET cLast
20:    READ
21:
22:    oThread:stop()
23:
```

```
24:    WAIT "Thread has stopped"
25: RETURN
```

The effect of this program is the same as in the previous example program: a series of three bitmaps is shown in the XbpStatic object created in line #9 and #10, while the user can enter data due to the READ command in line #20. The object representing the second thread is referenced in a LOCAL variable which enables us to start and stop the thread as desired.

The usage of an Animation object is very simple. It is created in line #12 where an array of bitmap file names is passed to the *:new()* method. The animation is started in line #15 and stopped in line #22. That means: we need to call only three (!) methods in a program for effectively using an Animation object. These methods are *:new()*, *:start()* and *:stop()*. What happens with the member variables and other methods shown in the table listing the Specification of an Animation class? Well, they are used internally by the Animation object and provide for its "intelligence". There is something going on behind the scenes when calling *:start()* and *:stop()*. You cannot see this in the example code, but you must understand it for the implementation of the Animation class:



When an Animation object executes the *:start()* method, it loads the bitmaps when thread B begins with executing program code or displaying bitmaps. The bitmaps are displayed in a round robin scheme while thread B is running and they are released before thread B terminates.

This circumscribes three vital stages of a thread that must be understood for successfully implementing user-defined Thread classes, such as the Animation class: a thread *starts*, it *runs* and it *ends*. The Thread class has three pre-defined methods that are to be used in user-defined Thread classes for the three stages:

:atStart() This method is called once. It is called automatically before a thread runs the *:execute()* method. An Animation object uses the *:atStart()* method to create XbpBitmap objects which actually load the bitmap files required for the animation.

:execute() This method may be called repeatedly as long as the thread runs. It is called automatically after *:atStart()* and implements the code which is executed in the new thread. An Animation object displays bitmaps in the *:execute()* method.

:atEnd() This method is is called once. It is called automatically before a thread terminates code execution. An Animation object uses the *:atEnd()* method to release XbpBitmap objects and their system resources.

Creating an Animation object

Up to now you have seen the specification of the Animation class, how objects of this class can be used within a program and how the program flow looks like when an Animation object is active. You also know that the Xbase^{++} Thread class has three methods which are called implicitly when a thread is started, while it is running and when it ends. With this knowledge it should be easy to understand the implementation of the Animation class:

```
01: CLASS Animation FROM Thread
02: PROTECTED:
03:
04:    // data required for displaying animated bitmaps
05:    VAR cDllName, aSource
06:    VAR aBitmaps, nCurrent, nTotal
07:    VAR aRect
08:
09:    // overloaded methods
10:    METHOD atStart, execute, atEnd
11:
12: EXPORTED:
13:    // overloaded method
14:    METHOD init
15:
16:    // new method in Animation class
17:    METHOD stop
18: ENDCLASS
19:
20:
```

```
21: METHOD Animation:init( aSource, cDllName )
22:    ::Thread:init()
23:    ::aSource  := AClone( aSource )
24:    ::cDllName := cDllName
25: RETURN self
```

The class declaration defines all instance variables an Animation object has in addition to the ones available in the Thread class. The methods *:atStart()*, *:execute()*, *:atEnd()* and *:init()* must be declared again, although they do exist in the super class. These methods are called implicitly and implement code that makes an Animation object different from a Thread object. The *init()* method is called implicitly within *:new()*. Just remember how an Animation object is created:

```
12:    oThread := Animation():new( { "Phase1.bmp", ;
13:                                  "Phase2.bmp", ;
14:                                  "Phase3.bmp"  } )
```

The same parameters passed to *:new()* are passed on to *:init()* and that's how an Animation object obtains the "knowledge" which bitmaps participate in the animation. When an array is passed to a method and stored in an instance variable, it is always a good idea to make a copy of that array (line #23). The most important part of the *:init()* method, however, is line #22 where the super class's *:init()* method is called. An Animation object cannot start a thread without initializing its super class, it would create a runtime error instead. But everything is fine here, the Animation object is properly initialized and ready to run the new thread.

Running the animation

The thread is started by calling the *:start()* method inherited from the Thread class. When this method is invoked the following code is executed in the new thread. It brings the animation to life and displays the bitmaps.

```
28: METHOD Animation:atStart( oXbp, nInterval, aRect )
29:    LOCAL i
30:
31:    IF nInterval == NIL
32:       nInterval := 10
33:    ENDIF
34:
35:    IF aRect == NIL
36:       aRect := oXbp:currentSize()
37:       aRect := { 0, 0, aRect[1], aRect[2] }
38:    ENDIF
39:
40:    ::aRect    := aRect
41:    ::nCurrent := 0
42:    ::nTotal   := Len( ::aSource )
```

```
43:    ::aBitmaps := Array( ::nTotal )
44:
45:    FOR i:=1 TO ::nTotal
46:       ::aBitmaps[i] := XbpBitmap():new():create()
47:       IF Valtype( ::aSource[i] ) == "N"
48:          ::aBitmaps[i]:load( ::cDllName, ::aSource[i] )
49:       ELSE
50:          ::aBitmaps[i]:loadFile( ::aSource[i] )
51:       ENDIF
52:    NEXT
53:
54:    ::setInterval( nInterval )
55: RETURN self
56:
57:
58: METHOD Animation:execute( oXbp )
69:    LOCAL oPS := oXbp:lockPS()
60:
61:    ::nCurrent ++
62:    IF ::nCurrent > ::nTotal
63:       ::nCurrent := 1
64:    ENDIF
65:
66:    ::aBitmaps[ ::nCurrent ]:draw( oPS, ::aRect )
67:    oXbp:unlockPS( oPS )
68: RETURN self
```

The *:atStart()* method provides for all resources required by the *:execute()* method to run successfully. The XbpBitmap objects, for example, are created in line #46 and load the bitmaps in turn so that they are available when *:execute()* draws the images in line #66. Also, the time interval for an automatic repetition of *:execute()* is set in line #54 when the thread begins to run. It defaults to 10 hundredths of a second (line #32) and causes the Animation object to "sleep" for this period of time before calling *:execute()* again. This program logic is the same as we have discussed in the previous section. The only difference is how the thread is started:

```
1st try:    oThread:start( "Animate", oXbp, aBitmaps )

2nd try:    oThread:start( , oXbp )
```

When using the built-in Thread class, the name of the procedure a Thread object excutes in the new thread is passed to the *:start()* method. This is not the case in user-defined Thread classes since the code for the new thread is implemented in the *:execute()* method, which is reserved for this very purpose. All but the first parameters received by the *:start()* method are passed also to *:atStart()*, *:execute()* and *:atEnd()*. The Animation object knows where to draw the bitmaps because the parameter *oXbp* references an XbpStatic object, or Xbase Part. This parameter is passed to *:execute()* each time the method is invoked. So this is something

to consider for the class design of user-defined Thread classes: We could store the Xbase Part in an instance variable but this is not necessary since it remains accessible via the parameter list. This approach has the additional advantage that there is less code to write for the clean-up at the end of a thread.

Memory issues

Cleaning up memory is good programming practice once you enter the multi-threading business. This is not an issue in single-threaded programs because the Xbase^{++} garbage collector does it automatically for you. But when you use multiple threads, you can relieve the garbage collector in its work and gain performance. A perfect time for doing a clean-up is when a thread ends and that is exactly what the *:atEnd()* method does:

```
71: METHOD Animation:atEnd( oXbp )
72:    AEval( ::aBitmaps, {|o| o:destroy() } )
73:    oXbp:invalidateRect( ::aRect )
74:    ::aBitmaps := NIL
75:    ::aRect    := NIL
76: RETURN self
```

In line #72 system resources allocated by XbpBitmap objects are released. Then the Xbase Part *oXbp* is informed that the rectangle on the screen occupied by the bitmaps is no longer valid (line #73) - which causes a screen update. Finally, all array references created in the *:atStart()* method are destroyed by assigning NIL to the corresponding instance variables.

Although memory issues are nothing you are bothered with in Xbase^{++} you should follow these rules when you define your own Thread classes:

1. Implement an *:atEnd()* method that does a clean-up.
2. Call the *:destroy()* method for all Xbase Parts that become obsolete or unaccessible (PROTECTED instance variables!) when the thread has ended.
3. Assign NIL to all instance variables that contain references to arrays, code blocks or objects when they become obsolete or unaccessible.

Stopping the animation

We have discussed now the code that runs in the thread created by an Animation object. The code implemented in the *:execute()* method is repeated forever unless we tell the object to terminate the thread. This is done in the method *:stop()* whose code is listed below:

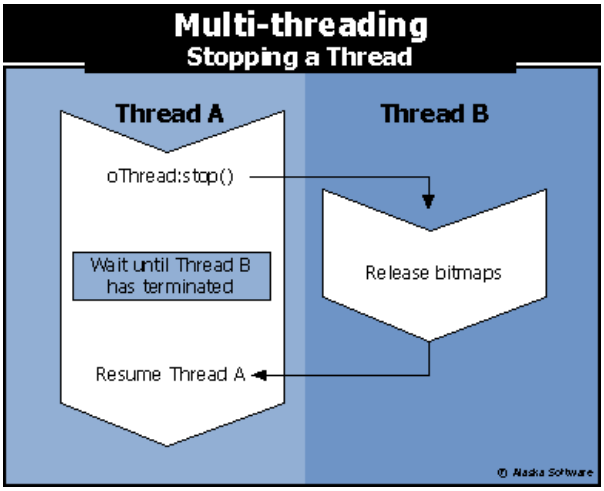
```
80: METHOD Animation:stop
81:    ::setInterval( NIL )
82:    ::synchronize( 0 )
83: RETURN self
```

The method consists effectively of two lines of code (line #81 and #82) which make it both simple to implement and hard to understand if you are not familiar with multi-threaded

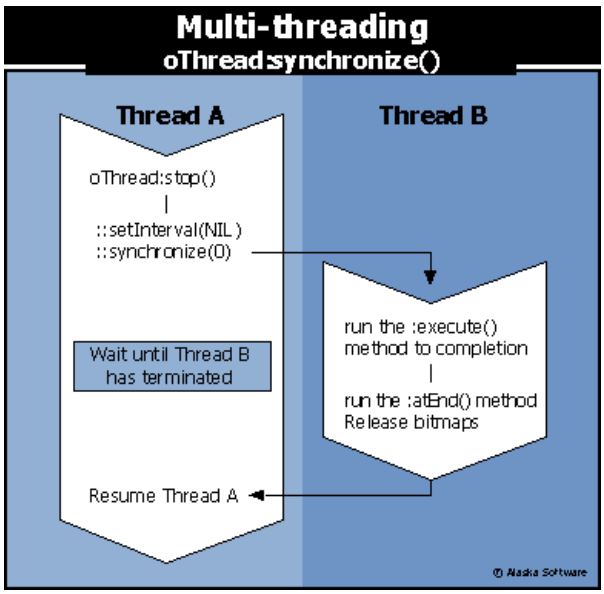
program logic. However, it is imperative for you to understand these two lines of code when you intend to implement your own Thread classes. To stop a thread is a vital issue in multi-threaded programs but you have absolutely no control over the time when a thread ends because this lies within the responsibility of the operating system. There are different strategies how to stop a thread and you have to implement different code depending on how your Thread class runs, better to say: how the *:execute()* method is implemented.

An Animation object relies on a time interval being set for repeated execution of the *:execute()* method and this makes it easy to stop the thread. When the time interval is set to NIL (line #81), the *:execute()* method is not called again and the thread ends.

We have already discussed that *:synchronize(0)* causes one thread to halt until another thread has ended. However, calling this method from within a user-defined Thread class makes line #82 really hard to digest and you ought to recall the situation when the *:stop()* method is executed. We have two threads running, A and B. Thread A knows the Animation object that represents thread B. Thread A stops thread B and must wait until thread B has actually ended:



The *:stop()* method is called in thread A which implies that the code of this method is also executed in thread A. From this it becomes obvious that *::synchronize(0)* (line #82) is executed in thread A and that it is thread A which is "synchronized" with thread B. During the synchronization, the *:execute()* method runs to completion and the *:atEnd()* method is executed before thread B finally ends:

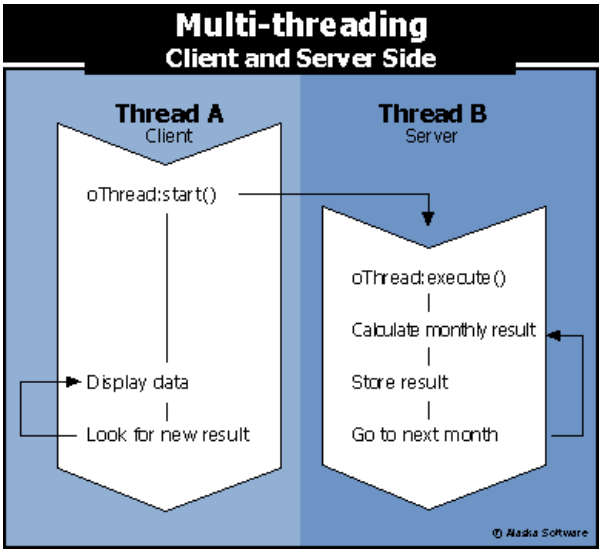


The point of this discussion is to make it clear that a Thread object will always have methods that are executed in two different threads. The *:stop()* method, for example, is a method that cannot be executed in the thread represented by the Animation object. Just think what would happen when *:stop()* would be called inside the *:execute()* method? The thread would be synchronized with itself, or, in other words, it would be busy with waiting for its own end, which is a deadlock situation. As a matter of fact, this would be a major logical programming error and Xbase^{++} prevents you from a deadlock by raising a runtime error if your program runs into this situation.

The two sides of a Thread object

It is not obvious from the implementation of a user-defined Thread class which methods are executed in which thread, unless you know the purpose of the three pre-defined methods *:atStart()*, *:execute()* and *:atEnd()*. Code that must run in the thread represented by the Thread object must be implemented in these methods. However, a Thread object has methods which must be called from another thread. This makes it necessary to distinguish two kinds of methods from a logical point of view which is most easily done using the terms *Client* and *Server*.

You may think of a Server being the black box in the computer room down the hallway and a Client being a similar thing sitting on your desk. This view is correct in the context of hardware. But in the field of software, Client/Server is a concept not necessarily related to physical entities. Instead, different parts of a software can play the Client/Server roles when the program is split into multiple threads. Assume a browser that displays a monthly consumer statistic. If a thread calculates consolidated data, a user can start viewing the results for the first months already while the thread still calculates the rest of the year. This would need two threads, one for the display and one for the calculation:



In this scenario, thread A uses data created by thread B, or, in other words, thread A is being served with data by thread B. Hence, it is legitimate to state that thread A is a Client of thread B while thread B is the Server thread. Looking at this from the Thread object's point of view we can conclude that *:start()* is a method which must be called in the Client thread because it causes the Server thread to run. Since a thread cannot be started twice while it is running, the *:start()* method is restricted to a Client thread, it is a client-side method.

In contrast, the *:execute()* method always runs in the thread represented by the Thread object. It is impossible that this method runs in another thread, it is a server-side method. We can, therefore, distinguish two sides of a Thread object by the thread in which methods are executed: client-side and server-side. The former consists of methods which cannot be executed in the thread represented by the object, while the latter is a group of methods that are always executed in this particular thread. There is, however, a third group of methods which can be executed in any thread. They define settings about *how* a thread is executed. This covers thread priority and timing issues.

Method groups of the Thread class

Client side	Server side	Settings
:new()	:atStart()	:setInterval()
:init()	:execute()	:setPriority()
:setStartTime()	:atEnd()	
:start()	:quit()	
:synchronize()		

The distinction beween client-method and server-method of a Thread object is useful since both terms help to structure a multi-threaded program easily. Server-methods perform tasks independently of the rest of a program (they serve something else), while client-methods create threads and control the program flow between them.

Basic programming techniques using threads

Thread usage and creating a user-defined Thread class is discussed in detail in the chapter Building an Animation class where a variety of program-logic related issues is outlined. This chapter deals with programming techniques and implementation issues relevant for applying multi-threading in your database applications.

Calculating statistics from databases

The calculation of a statistic is a common task in database applications and it provides a good example for a variety of issues relevant for multiple thread usage. In its broadest sense, the term "statistic" implies that a program has to pass through all records of a database and compute a result from its field values. For example, a question that can be answered by computing a sales statistic might be; "What is the gross revenue of last year?". Let us take this question as starting point in this discussion and look at a simple implementation:

```
01:   LOCAL nRevenue
02:
03:   USE Sales
04:
05:   SUM FIELD->SALES TO nRevenue ;
06:   FOR Year( FIELD->SALESDATE ) = Year( Date() ) - 1
07:
08:   CLOSE Sales
09:
10:   ? nRevenue
```

The code shows five major characteristics common to all statistics: we need variables to hold the result (line #1), a database is opened (line #3), the statistic is computed (lines #5 and #6), the database is closed (line #8) and the result is displayed (line #10). What the code does not show, but what is also a common feature of all statistics, is that the calculation can be very time consuming until the result can be displayed. Let us assume now that a user would like to do some other things while the statistic is being calculated in lines #5 and #6. There is no chance in the above implementation to accomplish this unless we use a thread that runs the time consuming part of the code.

Keep the term "time consuming" in mind. It is the key for identifying the code that must be isolated in order to be executed in a separate thread. In the example, it is the SUM command that takes time and we convert it to its functional equivalent before we run it in a thread (Note: the functional form of a command is easily obtained by compiling a PRG file using the /p switch and copy/paste the result from the PPO file into the PRG file). The modified example shown below does the same as before:

```
01:   LOCAL nRevenue, bEval, bFor
02:
03:   USE Sales
04:
05:   nRevenue := 0
06:   bEval    := {|| nRevenue += FIELD->SALES }
07:   bFor     := {|| Year( FIELD->SALESDATE ) = Year( Date() ) - 1}
08:
09:   DbEval( bEval, bFor )
10:
11:   CLOSE Sales
12:
13:   ? nRevenue
```

The statistic is defined in two code blocks that are passed to the DbEval() function (line #9). The function steps through the database and computes the result by evaluating the code blocks. That means: we have a function name (*DbEval*), two variables (*bEval, bFor*) used as parameters for the function, and this is all we need to run a thread. When we create a thread object, line #9 could run in the thread:

```
08:   oThread := Thread():new()
09:   oThread:start( "DbEval", bEval, bFor )
```

The DbEval() function is executed in the thread and gets two parameters passed by the thread object. This looks correct, but the code will bomb with the error message "Unknown/Invalid symbol for alias"! The reason why it will produce a runtime error is due to the fact that work areas are thread-local resources in Xbase^{++} (refer to The work space of Xbase^{++} in the Xbase^{++} documentation for details). Although the database is opened in line #3, its fields are not visible when the code blocks are evaluated in the second thread. The expressions FIELD->SALES and FIELD->SALESDATE lead to the error because the fields exist only in the current thread, not in the second one.

To resolve this problem, we have to open the database in the second thread before DbEval() is actually executed. The easiest way for this is by assigning two additional code blocks to the thread object:

```
01:   LOCAL nRevenue, bEval, bFor, oThread
02:
03:   USE Sales
04:
05:   nRevenue := 0
06:   bEval    := {|| nRevenue += FIELD->SALES }
07:   bFor     := {|| Year( FIELD->SALESDATE ) = Year( Date() ) - 1}
08:
09:   oThread         := Thread():new()
10:   oThread:atStart := {|| DbUseArea(,,"Sales") }
11:   oThread:atEnd   := {|| DbCloseArea() }
11:   oThread:start( "DbEval", bEval, bFor )
```

```
12:
13:   Browse()            // <... do something else ...>
14:
15:   CLOSE Sales
16:   oThread:synchronize(0)
17:   ? nRevenue
```

The code blocks assigned to the instance variables *:atStart* and *:atEnd* (lines #10 and #11) are evaluated by the Thread object when the thread starts and when it ends, respectively. They are perfect to use for tasks that are required only once in a thread, like opening and closing a database, for example.

The statistic is now calculated independently from any other code which means that a user can do something else while the thread computes the result. To demonstrate this in the example code, the function Browse() is called in line #13 which allows the user to view the Sales database while the thread iterates through the same file and computes the statistic. There is no possibility, however, that the current thread (*Browse()*) interferes with the second thread (*DbEval()*) when changing the record pointer of the database, since the file is opened twice (lines #3 and #10) and each thread maintains its own work area (thread-local). When the user is done with browsing, the result of the calculation is displayed after calling *:synchronize(0)* in line #16 (remember: always make sure a thread has ended!).

The example shows that you can execute basically any Xbase^{++} function in a thread without having to build a specialized Thread class. The DbEval() function is a very good candidate for a separate thread because it iterates through a database on its own (note: DbEval() is faster than a DO WHILE .NOT. Eof() loop). This, again, can be pretty time consuming, depending on how many records exist that must be processed. However, this does not really matter as long as the user can do other things and does not have to wait until DbEval() returns. Since the task for DbEval() is defined in a code block, you can perform calculations of any complexity using this approach. All you have to make sure is to open/close the necessary database(s) in the thread when it starts/ends.

Average and standard deviation

After we have seen how to calculate a simple statistic from a database in a separate thread, we will discuss now a more complex example. Normally, a single figure, like a total, is not sufficient to give a full picture, there are more things to look at. Average, percentage, total count or standard deviation are common figures in statistical analysis and they are quite useful for obtaining meaningful information from a set of data. Assume the question "How many cars did we sell in the first quarter of this year, what was the average price and what was the total revenue compared to the last quarter?".

To compute such figures fast, we must get as much information as possible by passing once through a database since skipping through a file is the "most expensive" operation. This implies that we have to program a function that computes multiple results by passing once through a database. Let us first see what this means in terms of code. The function programmed in the following example calculates the standard deviation, total count and total

sum for a numeric database field (Note: the standard deviation indicates how much single values deviate from the average of all values. A detailed description of the standard deviation goes beyond the scope of this discussion. It can be found in any good statistics book).

```
01: PROCEDURE Main
02:    LOCAL nStdDev, nTotal, nCount, bFor
03:
04:    USE Cars
05:
06:    bFor    := {|| Month(FIELD->SELLDATE) < 4 }
07:
08:    nStdDev := DbStdDev( "SELLPRICE", bFor, @nTotal, @nCount )
09:
10:    ? "# of cars sold:", nCount
11:    ? "Gross income  :", nTotal
12:    ? "Average price :", nTotal / nCount
13:    ? "Std. deviation:", nStdDev
14: RETURN
15:
16:
17: FUNCTION DbStdDev( cFieldName, bFor, nSum, nCnt )
18:    LOCAL nPos := Fieldpos( cFieldName )
19:    LOCAL nSqr := 0
20:
21:    nSum := 0
22:    nCnt := 0
23:
24:    DbEval( {|n| n := FieldGet( nPos ), ;
25:                  nSum += n           , ;
26:                  nSqr += n ^ 2       , ;
27:                  nCnt ++             ;
28:            }, bFor                     )
29:
30: RETURN Sqrt( ( (nCnt*nSqr) - (nSum^2) ) / ;
31:                (nCnt * (nCnt-1) ) )
```

The common single-threaded approach to calculate multiple results with one function is to pass parameters with the reference operator (@) to the function (line #8) where they are used for computation (lines #21-#27). The results are displayed after the function returns (line #10-#13). However, it is not possible to pass a parameter by reference to a function that is executed in a thread. This would mean to use the @ operator when calling the *:start()* method of a Thread object and in this case, the operator is ignored. The parameter would "arrive" in the thread but its value would remain unchanged in the calling routine when the thread ends.

There are two possibilities to make the function DbStdDev() "threadable". We could replace the LOCAL variables *nTotal* and *nCount* with a two element array. If it holds the value zero

in both elements, the array elements can be used for the calculations done in lines #25 and #26. In this case, function DbStdDev() would receive the array as third parameter and the fourth could be dropped.

Although this possibility is feasible, it is not optimal in terms of performance because accessing an array element is slightly slower than acessing a LOCAL variable. Since the access takes place within the DbEval() code block, the "slightly slower" can accumulate to a considerable loss in speed, depending on the size of the database. The better approach is, therefore, to embed the LOCAL variables in a code block and pass it to the *:start()* method. This is done in the following code which shows the modifications that allow the DbStdDev() function to be executed in a thread. Note the lines #6 and #36 when you read the code:

```
01: PROCEDURE Main
02:    LOCAL nStdDev, nTotal, nCount, bFor, bAssign, oThread
03:
04:    bFor            := {|| Month(FIELD->SELLDATE) < 4 }
05:
06:    bAssign         := {|n1,n2| nTotal:=n1, nCount:=n2 }
07:
08:    oThread         := Thread():new()
09:    oThread:atStart := {|| DbUseArea(,,"Cars") }
10:    oThread:atEnd   := {|| DbCloseArea() }
11:
12:    oThread:start( "DbStdDev", "SELLPRICE", bFor, bAssign )
13:    oThread:synchronize(0)
14:
15:    nStdDev         := oThread:result
16:
17:    ? "# of cars sold:", nCount
18:    ? "Gross income  :", nTotal
19:    ? "Average price :", nTotal / nCount
20:    ? "Std. deviation:", nStdDev
21: RETURN
22:
23:
24: FUNCTION DbStdDev( cFieldName, bFor, bAsgn )
25:    LOCAL nPos := Fieldpos( cFieldName )
26:    LOCAL nSqr := 0
27:    LOCAL nSum := 0
28:    LOCAL nCnt := 0
29:
30:    DbEval( {|n| n := FieldGet( nPos ), ;
31:                  nSum += n           , ;
32:                  nSqr += n ^ 2       , ;
33:                  nCnt ++             ;
34:            }, bFor                     )
```

```
35:
36:    Eval( bAsgn, nSum, nCnt )
37:
38: RETURN Sqrt( ( (nCnt*nSqr) - (nSum^2) ) / ;
39:                (nCnt * (nCnt-1) ) )
```

The code demonstrates an elegant solution to work around the shortcoming of the reference operator when a function is to run in a thread. The programming technique of embedding LOCAL variables in a code block can be used whenever LOCALs must be accessible outside the function where they are declared. By embedding the two LOCALs *nTotal* and *nCount* in line #6, they remain accessible **inside** the code block, and the values computed in DbStdDev() are assigned to the variables in line #36 where the code block is evaluated and receives the computed data as parameters.

Function DbStdDev() is very useful for statistical analysis since it computes three key figures (total count, total sum, standard deviation) that can be used to derive other key figures easily, like average or percentage, for example. During significant "number crunching" -which can be very time consuming- the user can do other things while the thread is running. As a matter of fact, you could insert whatever code you like between line #12 and #13 where the new thread is started and synchronized with the current thread. Note also line #15: the return value of DbStdDev() is obtained from the Thread object. It stores the return value of the function executed in the thread in its instance variable *:result*.

About threads and event loops

After we have seen how easy it is to calculate extensive statistics in a separate thread, or asynchronously, we have to look at ways how to present the results in a better way. Up to now, the examples have used the Qout() function (? command) to display the results which is not appropriate in a pure GUI application. When we choose an XbpDialog object as the application window, computed results are easily displayed using XbpStatic objects. Let us assume that a user can enter data in an XbpDialog window while a thread computes the statistic. When the thread is done, an additional window pops up and displays the result. That means, there is more than one thread and more than one window, or Xbase Part, involved.

If you plan to use Xbase Parts in more than one thread, there are two rules you have to keep in mind:

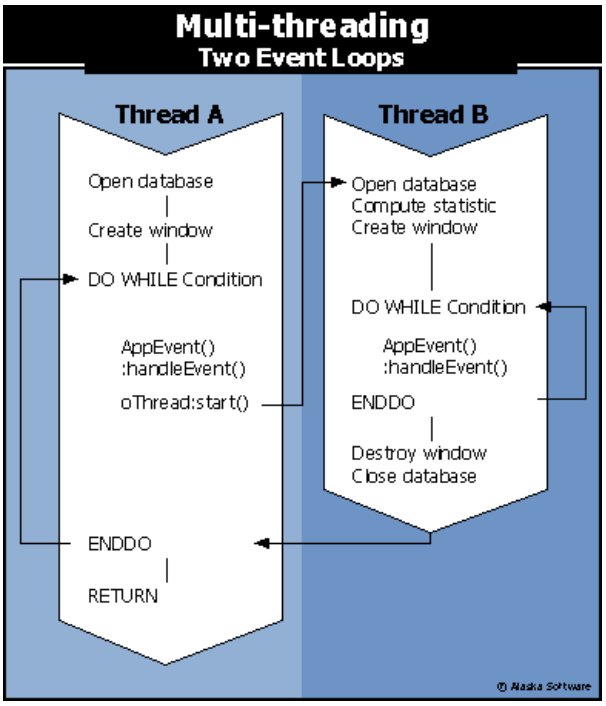
1. Each thread that creates Xbase Parts must run an event loop.
2. Xbase Parts receive events only in the thread that has created them.

Both rules are equally important when you design your programs to use multiple windows **and** multiple threads. If you don't comply with these rules, your application will not work. It will rather appear to you that some parts of your program work while others "hang", or don't do anything at all. However, there are two approaches you could follow: A) one thread computes the statistic and displays the result, or B) one thread displays all windows, another computes the statistic.

You can find working examples for both approaches in the directory \SAMPLES\DBSTDDEV and we will focus on the differences in program logic rather than discussing the entire code (most of it is required for the user interface which is not covered in this chapter).

One window per thread

The first approach is programmed in MAIN1.PRG where the Main procedure creates the application window and a thread is started via a pushbutton. The thread calculates the statistic and displays the results in a window when the database is entirely scanned. The following diagram illustrates the program flow (NOTE: Thread B stands for any number of threads that can be started from Thread A).

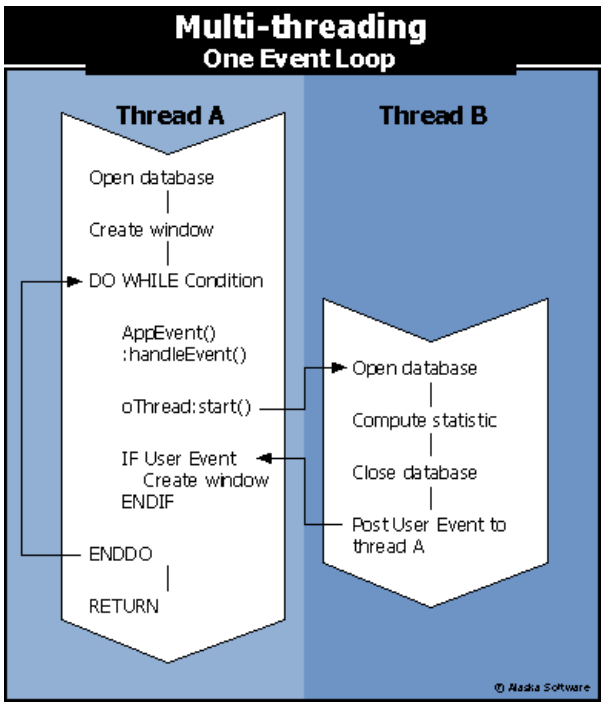


Thread A (the Main() procedure) opens a database, creates the application window and runs an event loop. When a pushbutton is clicked, thread B is started via the *:activate* code block (not indicated in the diagram). The new thread opens a database, computes a statistic, creates a window and runs an event loop. Thus, we have (at least) two event loops running parallel. The first addresses events to the window created in thread A and the second loop sends events to the window of thread B. Thread B ends when its event loop is exited, i.e. when the second window is closed. The program is terminated when the application window is closed because this exits the event loop in thread A and ends the Main() procedure.

Multiple windows in one thread

Opening multiple windows in one thread is a common situation and you have most likely done this already. However, we have to deal now with the situation that we do not know how long the calculation of the statistic will take. That means, we have to detect when the thread ends and only then should a new window pop up to display the results. We cannot create the new window in thread B because it would not receive events (remember rule #1).

In the example program MAIN2.PRG this problem is solved elegantly. Have a look at the next diagram, which shows the program flow of MAIN2.PRG (by the way, it does the same as MAIN1.PRG but uses a different programming technique):



The entire screen output is confined to thread A while thread B executes the invisible part of the program. As in the previous example, the new thread is started via *:activate* code block of a pushbutton. The thread does its "number crunching" task and posts a user-defined event to thread A, just before it ends. Well, to be exact, the event is not posted to thread A but to a window created in thread A: the application window. Let us see first how this looks in the code and discuss the implications afterwards (only the relevant parts are shown below):

```
01:   /*
02:    * Code running in thread A
03:    */
04:
05:    #define xbeUser_Eval     xbeP_User + 1
06:
```

```
07:    SetAppWindow( oDlg )
08:
09:    DO WHILE .T.
10:       nEvent := AppEvent( @mp1, @mp2, @oXbp )
11:       IF nEvent == xbeUser_Eval
12:          Eval( mp1 )
13:       ELSE
14:          oXbp:handleEvent( nEvent, mp1, mp2 )
15:       ENDIF
16:    ENDDO
17:
18:
19:   /*
20:    * Code executed at the end of thread B
21:    */
22:    bUser := {|| ResultWindow( <...> ) }
23:
24:    oThread:atEnd := ;
25:     {|| DbCloseArea() , ;
26:         PostAppEvent( xbeUser_Eval, bUser,, SetAppWindow() ) }
```

At first we define a new event constant in line #5 as a precondition for the program logic. This user-defined event constant must use xbeP_User as offset, otherwise it might interfere with events created by Xbase^{++}. The application window is made accessible for all threads in line #7 where it is passed to SetAppWindow(). Note that *oDlg* stands for an XbpDialog window created in thread A before the program enters the event loop in lines #9 to #16. This loop runs until the program ends.

Now, what happens when thread B is done with the calculation? The code to be executed at thread B's end is defined as code block that is assigned to the *:atEnd* instance variable so that it is executed automatically. Within the code block in line #26, the database is closed and the user-defined event constant is posted to SetAppWindow(). The PostAppEvent() function receives as second parameter the code block defined in line #22. It calls a function which creates the window displaying the results of the statistic. Because SetAppWindow() returns a window created in thread A, the event is retrieved from the event queue of thread A in line #10 and the code block *bUser* arrives in the first message parameter *mp1* of the AppEvent() function. This again closes the circle: the code block is evaluated in line #12 which causes the result window to be created in thread A and to receive events in thread A's event loop.

User-defined events provide a very powerful programming technique. They can be used to control a single-threaded application but they show their real strength in a multi-threaded program. When you know which thread an Xbase Part was created in, you can post events along with arbitrary values across thread boundaries, just by using a particular Xbase Part as addressee of the event (remember rule #2 stated at the beginning of this section).

Summary

Part I of the Multi-Threading Tutorial has discussed a variety of issues for using multiple threads in an Xbase^{++} program. These are in particular:

- Identifying tasks for separate threads
- Thread safeness means re-entrant code
- Starting and stopping a thread
- Implementing user-defined Thread classes
- Client-side and server-side methods of a Thread object
- Database usage in threads
- GUI event processing using event loops
- Cross-posting of user-defined events between threads

When you have worked through Part I of this tutorial, you have a pretty good idea of the DO's and DONT's in multi-threading and you are well prepared for Part II of this tutorial. It is going to be part of the next TechWire and we will discuss some example programs that demonstrate HOWTO's, or what can be done with threads in application programs. To give you an idea of what is coming, we include a "sneak preview" on the next page.

Happy Multi-Threading

Your Alaska-Software Team

Multi-Threading Tutorial Part II (preview)

Incremental search in browses

The programming technique of using two event loops in two threads allows for an easy implementation of a generic incremental search routine for a browser. All we need to do is to combine the various features of Xbase^{++} and "play the multi-threading piano" using a database, an XbpBrowse object and an XbpSLE object as the key components.

A database "watchdog"

There is an intrinsic problem in multi-user database programming arising from the fact that two users may change the same record of a database at the same time. This is known as "lost update" situation and there are a variety of strategies to cope with it. We will look at a solution to this problem that uses a thread which warns a user about a lost update while data is edited.

Combo boxes and database fields

Different problems may occur in the usage of combo boxes for editing database fields. We are going to discuss the implementation of a DbComboBox() class that makes a programmer's life easier. This class comes with an "autofill" feature that takes advantage of a thread and fills a combo box with unique field values stored in a database.

High-speed browsing of record subsets

The browse display of a subset of records stored in a database is one of the most challenging programming problems in database applications, and there are many approaches to achieve a high performance in creating a subset and accessing it with a browser. The DbSubset class is an elegant solution for this problem since it works on indexed and non-indexed databases and has the flexibility of the SET FILTER command. In short, the DbSubset class is a pretty sophisticated example for applied multi-threading technology.

A personal reminder

We are going to discuss in this section a new area of problems that have little or nothing to do with database programming but can easily be solved with a user-defined thread class. We will talk about time-controlled execution of code and persistency of Thread objects, i.e. you can determine the exact point in time when your computer should do something, you can store thread objects to disk or send them from one computer to another.