Synchronisation Toolkit Revisited

by Primoz Gabrijelcic

Here I am, back with my favourite theme: interprocess synchronisation and communication. If you are new to *The Delphi Magazine* I should warn you that this is the third article in an occasional series, so you may like to start reading at the beginning: check out Issues 86 and 88.

The plan for this article was to present an implementation of a shared memory pool: a mechanism that allows multiple data producers to send data to one data manipulator. In fact, I already had all the code and half the article completed when I found out that my solution doesn't always work. Everything was OK when both ends of the pool (producer and manipulator) were implemented in the same application or in two 'normal' applications. But if I tried to change one of those apps into an NT service, the shared pool stopped working.

While searching for the bug I found that actually I ran into two problems: `CreateMutex` called from the application was not able to open the same mutex created in the service, and `PostThreadMessage` was unable to send a message from the application to the service. To release the tension I must admit that both problems are fixed and new `TGpSharedMemoryPool` is working fine. I'll describe its innards in the next article.

This article is completely NT-related (and that includes Windows 2000 and XP). None of those problems apply to the 9x platform.

CreateMutex

So what is wrong with the `CreateMutex` function? Let's examine a simple scenario. We have two processes both calling `CreateMutex` with the same mutex name. Their intentions are obvious: they want to use this mutex to synchronise access to some shared resource.

When the first process calls `CreateMutex`, it will return a mutex handle and set the error code (`GetLastError`) to `NO_ERROR` as expected. Then the second process calls `CreateMutex`. That call will also return a mutex handle but the error code will be `ERROR_ALREADY_EXISTS`. That way our process knows that the mutex was already created and that the kernel was smart enough to open that mutex instead of creating it.

That's fine, but what if one of those processes is actually a service? In that case, only the first process can access the mutex. The second one will get a null mutex handle and the error code `ERROR_ACCESS_DENIED`. That is quite confusing, as this error is not mentioned in the `CreateMutex` documentation. Still, a quick search in MSDN found that this is intentional. There is a boundary between services and normal applications that kernel objects (mutexes, events, files, etc) cannot cross without special preparation. MSDN states that 'Most services are installed in the local system account and run with special security rights as a result. Processes running in the local system account grant `GENERIC_ALL` access to other processes running in the local system, and `READ_CONTROL`, `GENERIC_EXECUTE`, and `GENERIC_READ` access to members of the Administrators group. All other access to the object by any other users or groups is denied. All calls to `CreateMutex` implicitly request `MUTEX_ALL_ACCESS` for the object in question. An interactive user does not have the rights required to obtain a handle to an object created from the local system security context as a result.'

So this is a security question. As such, there should be a way to work around it. MSDN continues: 'There are several solutions to this problem: You can set the security descriptor in your call to `CreateMutex` to contain a NULL DACL. An object with NULL-DACL security grants all access to everyone, regardless of security context... A second option would be to create a security descriptor that explicitly grants the necessary rights to the built-in group: Everyone. In the case of a mutex, this would be `MUTEX_ALL_ACCESS`. This is preferable because it will not allow malicious (or buggy) software to affect other software's access to the object.'

Gosh, what a mess. Security descriptor, DACL, security context... I must admit that I've spent quite some time looking for an answer, which I found in the Borland newsgroups (thanks go to the Tamarack Associates search engine at www.tamarack.com). Then I struggled to understand the answer and in the end I tried to find the missing information in MSDN. Finally I tried to remove all the unimportant stuff and reduce my collected material to a few pages in this magazine.

A Crash Course In NT Security

A one line solution to the `CreateMutex` problem is to use the `lpMutexAttributes` parameter. Now everyone will jump to the Delphi help and return unsatisfied: all that the help has to say is: 'The `lpSecurityDescriptor` member of the structure specifies a security descriptor for the new mutex. If `lpMutexAttributes` is `NULL`, the mutex gets a default security descriptor.' But what does the `lpSecurityDescriptor` member point to and how can we create this structure?

Before we get to the answer, I must point out that most of the API functions we need are not declared in the `Windows` unit, at least not in the one that is included with Delphi 5 (which I'm still depending on). Therefore, I have used a much better API translation, written by Marcel van Brakel. You can get it at www.delphi-jedi.org (click on the *API Library* link on the left, then find Win32API.zip).

Let us first get some overview of NT security. When a user logs to the NT, the system authenticates the user's account name and password. If the logon is successful, the system creates an *access token* (see the sidebar *A Little Security Dictionary*). It contains a *security identifier* that identifies the user's account and any groups to which the user belongs. Every process executed on behalf of this user will carry a copy of this access token.

The token also contains a list of the privileges held by the user (or the groups it belongs to), but we will skip that part.

More important is the other aspect of NT security, which is not associated with the logon session, but with the kernel object itself. When a securable object is created, the system assigns it a *security descriptor* that contains security information specified by the object's creator (or default security information if none is specified). What we usually do is pass a nil pointer for the `lpMutexAttributes` parameter (or other parameter of the `PSecurityAttributes` type) implying that we want the kernel to assign default security information to this mutex. But we already knew that. The real problem lies in the security descriptor: what it contains and how we can build one.

A security descriptor contains three things. It identifies the object's owner, optionally contains a DACL, and (also optionally) a SACL. To deal with the least important first, SACL (meaning *system access control list*) is a list that controls how the system audits (logs) attempts to access the object. We won't use it.

The other control list, *discretionary access control list* (DACL) is the one we have to create. It identifies the users and groups allowed or denied access to the object. DACL and SACL have both the same structure, they contain a list of ACEs: *access control entries*.

ACE is a basic security element and specifies a set of access rights, together with a *trustee* for whom the rights are allowed, denied, or audited. A trustee can be a user account, a group account (eg `Everyone`), or a logon session. A trustee is identified by a unique string, a security identifier (SID).

Definitions, definitions. Let's see how all this is used in action. When a thread tries to access an object, the system (that is, the kernel) looks at the DACL. If the object doesn't have a DACL, the system grants or denies access based on other information (logon session, for example). Otherwise, the system looks for ACEs in the object's DACL that apply to the thread. The system compares the trustee in each ACE to the trustees

A Little Security Dictionary

Absolute Security Descriptor
A security descriptor structure that contains pointers to the security information associated with an object.

Access Control Entry (ACE)
An entry in an access control list (ACL). An ACE contains a set of access rights and a security identifier (SID) that identifies a trustee for whom the rights are allowed, denied, or audited.

Access Control List (ACL)
A list of security protections that applies to an object. (An object can be a file, process, event, or anything else having a security descriptor.) An entry in an access control list (ACL) is an access control entry (ACE). There are two types of access control list, discretionary and system.

Access Token
An access token contains the security information for a logon session. The system creates an access token when a user logs on, and every process executed on behalf of the user has a copy of the token. The token identifies the user, the user's groups, and the user's privileges. The system uses the token to control access to securable objects and to control the ability of the user to perform various system-related operations on the local computer. There are two kinds of access token, primary and impersonation.

Discretionary Access Control List (DACL)
An access control list that is controlled by the owner of an object and that specifies the access particular users or groups can have to the object.

Locally Unique Identifier (LUID)
A 64-bit value guaranteed to be unique on the operating system that generated it (until the system is restarted).

Logon Identifier
An LUID that identifies a logon session. A logon ID is valid until the user logs off. A logon ID is unique while the computer is running; no other logon session will have the same logon ID. However, the set of possible logon IDs is reset when the computer starts up.

Logon Session
A logon session begins whenever a user logs on to a computer. All processes in a logon session have the same primary access token. The access token contains information about the security context of the logon session, including the user's SID, the logon identifier, and the logon SID.

Logon SID
A security identifier (SID) that identifies a logon session. You can use the logon SID in a DACL to control access during a logon session. A logon SID is valid until the user logs off. A logon SID is unique while the computer is running; no other logon session will have the same logon SID. However, the set of possible logon SIDs is reset when the computer starts up.

Security Descriptor
A structure and associated data that contains the security information for a securable object. A security descriptor identifies the object's owner and primary group. It can also contain a DACL that controls access to the object, and a SACL that controls the logging of attempts to access the object.

Security Identifier (SID)
A structure of variable length that uniquely identifies a user or group on all Windows NT implementations.

Self-Relative Security Descriptor
A security descriptor that stores all its security information in a contiguous block of memory.

System Access Control List (SACL)
An ACL that controls the generation of audit messages for attempts to access a securable object. The ability to get or set an object's SACL is controlled by a privilege typically held only by system administrators.

Trustee
In Microsoft Windows NT security, a trustee is the user account, group account, or logon session to which an access control entry (ACE) applies. Each ACE in an access control list (ACL) applies to one trustee.

identified in the thread's access token.

Why a thread and not a process? Because a thread may have different rights than a process or event different trustee, if it impersonates another user. No, I don't intend to discuss impersonation in more detail: that will have to wait for another occasion.

The system examines each ACE in sequence until one of the following occurs. An ACE may explicitly deny any of the requested access rights to one of the trustees listed in the thread's access token. In that case, the request is denied. If that doesn't happen, one or more ACEs may explicitly grant all the requested rights. In that case, the request is allowed. If all ACEs have been checked and there is still at least one requested access right that has not been explicitly allowed, access is denied.

This procedure clearly indicates that the order of ACEs in the DACL list is important. If there are ACEs that explicitly allow given rights but are preceded by an access-denying ACE, access will be denied because the denying ACE will be found first.

If the DACL field in the object's security descriptor is set to `nil`, a null DACL is created. A null DACL grants full access to any user that requests it: normal security checking is not performed for the object (except that the null DACL still doesn't give a normal application access to an object created in the `LocalSystem` account). A null DACL should not be confused with an empty DACL. An empty DACL is a properly allocated and initialised DACL containing no ACEs. An empty DACL denies any access to the object it is assigned to.

```
constructor TGpSecurityAttributes.AllowSID(sid: PSID);
var
  daclSize, sidSize : integer;
begin
  if Win32Platform <> VER_PLATFORM_WIN32_NT then Exit;
  // copy SID to internal field
  sidSize := GetLengthSid(sid);
  gsaSid := AllocMem(sidSize);
  Move(sid^, gsaSid^, sidSize);
  // create a dacl and add the SID, granting full access
  daclSize := SizeOf(ACL) + SizeOf(ACCESS_ALLOWED_ACE) + GetLengthSid(gsaSid);
  gsaDacl := AllocMem(daclSize);
  Win32Check(InitializeAcl(gsaDacl, daclSize, ACL_REVISION));
  Win32Check(AddAccessAllowedAce(gsaDacl, ACL_REVISION, GENERIC_ALL, gsaSid));
  // create a security descriptor and set the dacl
  Win32Check(InitializeSecurityDescriptor(@gsaSecDescr,
    SECURITY_DESCRIPTOR_REVISION));
  Win32Check(SetSecurityDescriptorDacl(@gsaSecDescr, true, gsaDacl, false));
  // initialize a security attribute
  FillChar(gsaSecAttr, SizeOf(gsaSecAttr), 0);
  gsaSecAttr.nLength := SizeOf(gsaSecAttr);
  gsaSecAttr.lpSecurityDescriptor := @gsaSecDescr;
  gsaSecAttr.bInheritHandle := false;
end; { TGpSecurityAttributes.AllowSID }
```

➤ *Listing 1: Creating a SID-allowing DACL.*

Please Let Me In

Finally, the time has come to show some code. This time, although it is not usually my practice, I'll use the bottom-up approach. First I'll show you the actual DACL-creating code and then I'll put it into the larger context. Somehow, things make more sense this way, mostly because the NT security model is full of small idiosyncrasies that must be taken into the account when designing the application.

The code (which is actually a constructor of a helper class that I'll describe later) is shown in Listing 1. It creates a DACL that allows full access for one SID. More details on how to get this SID will follow.

After checking that our system supports NT security, the code creates a copy of the constructor parameter. The logic behind this is that the access-allowing ACE in the DACL won't contain this parameter but only a pointer to it. As we will need the DACL later, after the constructor has finished the work and the parameter has long gone to parameter heaven, we must create a more persistent copy. The `gsaSid` variable is part of the `TGpSecurityAttributes` class and is only destroyed in its destructor with a simple `FreeMem`. The `GetLengthSid` function returns the size of the SID.

Next, the code must first provide a place for the DACL, a large enough memory buffer. To calculate its size, the code (in the line starting with `daclSize :=`) adds together the size of the standard ACL header (`SizeOf(ACL)`), the size of the access-allowing ACE structure (`SizeOf(ACCESS_ALLOWED_ACE)`) and the size of the SID. After that, the memory is allocated. Because the security descriptor we are building will only contain a pointer to the DACL memory, this memory buffer will only be freed in the class destructor.

Instead of manipulating this block of memory directly, `InitializeAcl` is used. It takes three parameters: a pointer to the DACL memory, the size of this memory block, and a third parameter, which must be set to the constant `ACL_REVISION`. By the way, you should really read the description of this function in the help as it contains the formula for calculating the DACL size (an example of

➤ *Listing 2: Creating an account-allowing DACL.*

```
constructor TGpSecurityAttributes.AllowAccount(const accountName: string);
var
  domain    : string;
  domainSize: DWORD;
  sid       : PSID;
  sidSize   : DWORD;
  use       : DWORD;
begin
  if Win32Platform <> VER_PLATFORM_WIN32_NT then Exit;
  // get the SID for the account name
  domainSize := 0;
  LookupAccountName(nil, PChar(accountName), nil, sidSize, nil, domainSize, use);
  sid := AllocMem(sidSize);
  try
    SetLength(domain, domainSize);
    Win32Check(LookupAccountName(nil, PChar(accountName), sid, sidSize,
      PChar(domain), domainSize, use));
    AllowSID(sid);
  finally FreeMem(sid); end;
end; { TGpSecurityAttributes.AllowAccount }
```

which you have seen in the previous paragraph).

To this DACL, which is currently initialised but empty (therefore denying any access to the object), we must add an access-allowing ACE. Unsurprisingly, `AddAccessAllowedAce` function does the trick. It takes four parameters: first a pointer to the DACL buffer, second `ACL_REVISION`, the third specifies the access rights that are granted and the last contains the address of the SID structure to which rights are granted.

Now we have a DACL, but no security descriptor. First we have to initialise it with the `InitializeSecurityDescriptor` function. That is fairly trivial: we provide it with a pointer to the security descriptor and a descriptor revision (which is always `SECURITY_DESCRIPTOR_REVISION`). Note that the `gsaSecDescr` field is not a pointer but a record and therefore we have to get its address with the `@` operator.

At last, we can assign the DACL to this security descriptor with the `SetSecurityDescriptorDacl` call. This takes a pointer to the security descriptor, a flag specifying that the DACL is present, a pointer to the DACL's memory, and a flag specifying that we don't want the default DACL.

Nearly done: now we only need to set up a security attribute structure (`TSecurityAttribute`). At least it doesn't require any weird security API calls. The code simply initializes it to all zeros, sets the length of the structure and sets the pointer to the security descriptor.

The last mystery we have to solve is how to get a SID. Well, it depends. I'll show the code to get an account SID and the code that retrieves the well-known group SID, but for all other cases you should dig into MSDN.

The `AllowAccount` constructor (Listing 2) takes an account name, converts it into the corresponding SID and calls the `AllowSID` constructor that we already know. There is only one function we need, `LookupAccountName`, but it has to be called twice. The first call will return the SID size, so we can allocate the appropriate amount of memory, and the second call will return the SID.

`LookupAccountName` takes seven parameters. The first can contain the name of the machine we want to look up the account on. If it is `nil` (it will be in our case) the account is looked up on the local system. The second parameter contains the name of the account we are interested in. The third parameter is the pointer to the SID: in the first call the code sets it to `nil`, indicating that it is only interested in the SID size (which is returned in the fourth parameter). Between the calls the SID buffer is allocated and in the second call the pointer to this buffer is passed in the third parameter.

Although we are not interested in the domain name of this account, we must still set the buffer for it aside or the second call to `LookupAccountName` will fail. In the first call the code sets the domain buffer parameter (parameter number five) to `nil` and domain size (the sixth parameter) to `0`. The first call sets the domain size, the code allocates appropriate buffer (by calling `SetLength`) and the second call returns the domain name.

The last parameter returns the type of the account, which we are not interested in.

That covers the users, but what about the user groups? Well, if we know the group name, we can use the same code (`AllowAccount`). That is, however, not advisable if we want to allow access to a certain well-known group (for example, `Everyone`), as its name may be translated in the localised Windows. The constructor in Listing 3, `AllowEveryone`, allows access to `Everyone` in a manner independent of the locale by building the SID for `Everyone` piece by piece. The code initialises the SID, sets its `Authority` field to a well-known value representing the `Everyone` group and then sets its first `SubAuthority` field to another well-known value representing the same group. As the internal structure of a SID is well beyond the scope of this article I won't explain this constructor in more detail. For other well known group SIDs see the `JwaWinNT` unit (part of the Jedi Win32 API translation). For help on the SID structure and other specifics, see MSDN.

I should also point out that there are simpler ways of constructing security descriptors. If you don't need NT 4.0 compatibility, explore the `ConvertStringSecurityDescriptorToSecurityDescriptor` function, which takes a textual representation of a security descriptor and converts it to a real descriptor. Very cool, very simple (once you know the input string format), but not appropriate if you must maintain the compatibility with NT 4.0 (as some of us still have to).

This covers all the important parts of the `TGpSecurityAttributes` class (see Listing 4). You have seen everything except the destructor (which only deallocates the memory) and the `GetSA` getter, which returns `nil` on the Windows 9x architecture and an address of the `gsaSecAttr` field on the NT architecture.

To use it in a real application, follow the pattern in Listing 5. The code should create the security attributes class and then use its `SecurityAttributes` property instead of `nil` when creating a

➤ *Listing 3: Creating an Everyone-allowing DACL.*

```
constructor TGpSecurityAttributes.AllowEveryone;
var
  siaWorld: SID_IDENTIFIER_AUTHORITY;
  sid     : PSID;
begin
  if Win32Platform <> VER_PLATFORM_WIN32_NT then Exit;
  // get the well-known Everyone SID
  siaWorld := SECURITY_WORLD_SID_AUTHORITY;
  sid := AllocMem(GetSidLengthRequired(1));
  try
    Win32Check(InitializeSid(sid, @siaWorld, 1));
    PDWORD(GetSidSubAuthority(sid, 0))^ := SECURITY_WORLD_RID;
    AllowSID(sid);
  finally FreeMem(sid); end;
end; { TGpSecurityAttributes.AllowEveryone }
```

securable object (an event in this case).

To make the code simpler, I have created wrapper functions that cover most of the expected usage (Listing 6): creating events, file mappings, mutexes and semaphores with full access (`AllowEveryone`) or named access (`AllowAccount`). In fact, the code in Listing 5 is taken from the `CreateEvent_AllowAccount` function.

My initial problem can now be solved by using `CreateMutex_AllowEveryone` instead of `CreateMutex`.

All this code is packed into the `GpSecurity` unit (included with this months code). I have also provided a fixed `GpSync` (described in the first article in the series) and `GpSharedMemory` (from the second article) where all securable objects are now allocated with the `_AllowEveryone` function. That means that you can use the new `GpSync` to synchronise a service with an application and the new `GpSharedMemory` to exchange data between a service and an app.

PostThreadMessage

If you still remember (after all this security-related talk), the second problem I encountered was that I couldn't send a message to the service. I was trying to accomplish that with the `PostThreadMessage`, but it was failing with the status `ERROR_INVALID_THREAD_ID`. Again, I found the answer in MSDN: 'Windows 2000/XP: This thread must either belong to the same desktop as the calling thread or to a process with the same *locally unique identifier (LUID)*. Otherwise, the function fails and returns `ERROR_INVALID_THREAD_ID`.' In other words, the sender and receiver must either belong to the same desktop or to the same logon session. The latter condition is never true for services, as every service has a separate logon session. We can satisfy the former condition by changing the service into an interactive service. That change alone makes `PostThreadMessage` work again.

Sometimes an interactive service is not an option and we must use a `PostThreadMessage` replacement. My solution was to create a messaging class that doesn't use Windows messages. The general idea is simple: create a shared memory buffer to hold messages and an event which signals that there are new messages in the memory buffer.

Listing 7 shows the public part of the 'messaging without messages' classes. `TGpMessageQueue` is an abstract base class that defines four different ways to post a message (more on that later) and a property to hold the message queue name. In a typical `GpSync` manner, all `Post` methods (and all the `Get` and `Peek` methods we'll soon encounter) take a timeout parameter. If the message queue can't be accessed in that amount of time, the function returns an error.

`TGpMessageQueueWriter` is a simple descendant that can only write to the message queue. It defines a constructor (taking the message queue name and the size of the message queue buffer) and a function, `IsReaderAlive`, which checks if the other part of the message queue, the message queue reader, is running. I should add that having a live message queue reader is not a requirement. A writer can post a message to the queue even if no reader is alive (but doing that on a regular basis would quickly fill up the memory buffer). There is no limitation on the number of concurrent writers: each message queue can have as many writers as needed.

In contrast to that, there can only be one reader per queue, `TGpMessageQueueReader`. This class defines two constructors, four `GetMessage` methods (corresponding to the four `PostMessage` methods in `TGpMessageQueue`) and

```
TGpSecurityAttributes = class
private
  gsaDacl    : PACL;
  gsaSecAttr : TSecurityAttributes;
  gsaSecDescr: TSecurityDescriptor;
  gsaSid     : PSID;
protected
  function GetSA: LPSECURITY_ATTRIBUTES;
public
  constructor AllowAccount(const accountName: string);
  constructor AllowEveryone;
  constructor AllowSID(sid: PSID);
  destructor  Destroy; override;
  property SecurityAttributes: LPSECURITY_ATTRIBUTES read GetSA;
end; { TGpSecurityAttributes }
```

➤ *Listing 4: Security attributes generator.*

```
var gsa: TGpSecurityAttributes;
begin
  gsa := TGpSecurityAttributes.AllowAccount(accountName);
  try
    Result := CreateEvent(gsa.SecurityAttributes, manualReset,
      initialState, PChar(eventName));
  finally FreeAndNil(gsa); end;
end;
```

➤ *Listing 5: Using the security attributes generator.*

➤ *Listing 6: Insecure object creators.*

```
function CreateEvent_AllowAccount(const accountName: string;
  manualReset, initialState: boolean; const eventName: string): THandle;
function CreateEvent_AllowEveryone(manualReset, initialState: boolean;
  const eventName: string): THandle;
function CreateFileMapping_AllowAccount(const accountName: string;
  hFile: THandle; flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD;
  const fileMappingName: string): THandle;
function CreateFileMapping_AllowEveryone(hFile: THandle; flProtect,
  dwMaximumSizeHigh, dwMaximumSizeLow: DWORD;
  const fileMappingName: string): THandle;
function CreateMutex_AllowAccount(const accountName: string;
  initialOwner: boolean; const mutexName: string): THandle;
function CreateMutex_AllowEveryone(initialOwner: boolean;
  const mutexName: string): THandle;
function CreateSemaphore_AllowAccount(const accountName: string;
  initialCount, maximumCount: longint; const semaphoreName: string): THandle;
function CreateSemaphore_AllowEveryone(initialCount, maximumCount: longint;
  const semaphoreName: string): THandle;
```

```
TGpMessageQueue = class
  destructor  Destroy; override;
  function PostMessage(timeout: DWORD; const msgData:
    string): TGpMQPostStatus; overload;
  function PostMessage(timeout: DWORD; flags:
    TGpMQMessageFlags; msg: UINT;
    wParam: WPARAM; lParam: LPARAM; const msgData: string):
      TGpMQPostStatus; overload;
  function PostMessage(timeout: DWORD; msg: UINT;
    const msgData: string): TGpMQPostStatus; overload;
  function PostMessage(timeout: DWORD; msg: UINT;
    wParam: WPARAM; lParam: LPARAM): TGpMQPostStatus;
      overload;
  property Name: string read mqName;
end; { TGpMessageQueue }
TGpMessageQueueWriter = class(TGpMessageQueue)
  constructor Create(messageQueueName: string;
    messageQueueSize: cardinal); override;
  function  IsReaderAlive: boolean;
end; { TGpMessageQueueWriter }
TGpMessageQueueReader = class(TGpMessageQueue)
  constructor Create(messageQueueName: string;
    messageQueueSize: cardinal;
    newMessageEvent: THandle{CreateEvent}); reintroduce;
      overload;
  constructor Create(messageQueueName: string;
    messageQueueSize: cardinal;
    newMessageWindowHandle: HWND; newMessageMessage: UINT);
      reintroduce; overload;
  function GetMessage(timeout: DWORD; var flags:
    TGpMQMessageFlags;
    var msg: UINT; var wParam: WPARAM; var lParam: LPARAM;
    var msgData: string): TGpMQGetStatus; overload;
  function GetMessage(timeout: DWORD; var msg: UINT;
    var msgData: string): TGpMQGetStatus; overload;
  function GetMessage(timeout: DWORD; var msg: UINT;
    var wParam: WPARAM; var lParam: LPARAM): TGpMQGetStatus;
    overload;
  function GetMessage(timeout: DWORD; var msgData: string):
    TGpMQGetStatus; overload;
  function PeekMessage(timeout: DWORD; var flags:
    TGpMQMessageFlags; var msg: UINT; var wParam: WPARAM;
    var lParam: LPARAM; var msgData: string):
    TGpMQGetStatus; overload;
  function PeekMessage(timeout: DWORD; var msg: UINT;
    var msgData: string): TGpMQGetStatus; overload;
  function PeekMessage(timeout: DWORD; var msg: UINT;
    var wParam: WPARAM; var lParam: LPARAM): TGpMQGetStatus;
    overload;
  function PeekMessage(timeout: DWORD; var msgData: string):
    TGpMQGetStatus; overload;
end; { TGpMessageQueueReader }
```

➤ *Listing 7: Message queue interface.*

four `PeekMessage` methods. Being a `TGpMessageQueue` descendant, the reader can also post messages back to its own queue, which can sometimes prove useful.

Why this abundance of messages? Well, because I went slightly over the top during the design phase. First, I wanted to have a `PostMessage` that directly mimics the existing `PostMessage`, that is I wanted to have `msg`, `wParam` and `lParam` fields. This is useful if you want to plug `TGpMessageQueue` into an existing design, which already uses `Post(Thread)Message` to send messages (as I had to). Next, I added a 'light' version that sends only a string: very nice if you are sending string messages with rich content (like XML), and another version that sends a message number and a string. To top all that, I have added a 'rule them all' version that contains all four parameters (message number, `wParam`, `lParam` and message string) plus a fifth field specifying which of the other four parameters are valid. Of course, each `Post` variant is accompanied by a `Get` variant (which retrieves the message from the queue) and a `Peek` variant (which retrieves the message contents but leaves the message in the queue). Internally, the first three variants are implemented by calling the five-parameter monster.

On the plus side you don't have to remember all that. Just choose the variant that suits your reqirements and use it.

```
procedure TGpMessageQueueReaderThread.Execute;
var
  awaited: DWORD;
  handles: array [0..1] of THandle;
begin
  handles[0] := mqrtNewMessageEvt;
  handles[1] := mqrtTerminateEvt;
  while true do begin
    awaited := WaitForMultipleObjects(2, @handles, false, INFINITE);
    if awaited <> WAIT_OBJECT_0 then
      break; //while
    if mqrtNotifyEvent <> 0 then begin
      SetEvent(mqrtNotifyEvent);
    end;
    if mqrtNotifyWindowHandle <> 0 then begin
      PostMessage(mqrtNotifyWindowHandle, mqrtNotifyMessage,
        WPARAM(mqrtParent), 0);
    end;
  end; //while
end; { TGpMessageQueueReaderThread.Execute }
```

➤ *Listing 8: Message queue reader thread.*

The reason for two `TGpMessageQueueReader` constructors is more sound. There are two different versions covering two different scenarios. If you use the message queue in the thread, then you should use the constructor that takes an event handle. When a new message arrives, the message queue reader (actually its working thread: see the code in Listing 8) will set this event. Your thread should wait on this event (probably in combination with some other waitable objects) and when it is set it should read all waiting messages.

When using the message queue reader in an event-driven architecture (for example, in the user interface thread), you should use the other constructor, which takes a window handle and a message number. When a new message arrives, the reader's working thread will post this message to the specified window handle.

This second approach is used in the `TGpMessageQueueReaderComp`, a component wrapper for the `TGpMessageQueueRead`. This component creates a hidden window and instructs the reader to post a special message to this window. When that message is processed by the window's message-processing code, it is remapped to a call to the component event handler.

To see how the message sending and receiving can be used in your program, see the demonstration program testGpSync, which is, together with the new `GpSecurity` unit and updated `GpSync` and `GpSharedMemory` units, part of this month's code download.

Primoz Gabrijelcic is the R&D Manager of FAB doo in Slovenia. You can contact him at gp@fab-online.com