My Data Is Your Data

The second part of a series of articles on inter-process synchronisation

Primoz Gabrijelcic

There is no multiprocess solution without data sharing. If you have to do something useful and need more than one application to do it, you need a way to share data between those applications. The standard answers to data sharing are files, databases, shared memory and specialised servers. In this article I will deal with the shared memory solution to the problem. We'll review the Windows mechanisms that allow us to share memory between processes, and then build a wrapper around those mechanisms.

This article follows on from *A Synchronisation Toolkit* in Issue 86. If you skipped the first part, don't worry: this month's source contains all the necessary code, including that from the first article.

There are plenty of ways in which processes can share and exchange data in Windows: files, pipes, mailboxes, sockets, etc. They are all exposed through the Win32 API. The most basic of all these tools is (at least for us old-timers who grew up on DOS) a simple piece of memory that two (or more) processes can access. A *shared memory* area is easy to understand and easy to use. The only trouble with the shared memory approach on Windows is that Win32 is rather lacking in support for it.

Windows 95 onwards implements shared memory in a same way, using memory-mapped files. Memory- mapping is an interesting concept that allows us to work with a file as if it was a part of the process memory. This allows us to use standard functions that work on a memory block and not on a file. Also, we can skip the loading of a file into memory buffers, as Windows does that for us.

So how does that help us share data between processes? Simple: two (or more) processes can map the same file. Each change a process makes is instantly visible in all the other processes. This is a simple consequence of the fact that all processes actually work with the same physical memory block where data from the mapped file is stored.

This is all well and good, and suitable for file processing, but does not really help us develop a shared memory solution. We certainly don't want to create a new file for each shared memory block. There is, however, a different way in which file mapping can be used. Instead of using a specific file, we can tell Windows that it should use the system page file for that purpose. In most cases, the page file won't even be used: if there is enough memory, page file memory mapping will be kept in the RAM. The contents of the mapping will only be stored to disk if there is a shortage of memory.

A Win32 API Way

A simple demonstration of shared memory access using only Windows API functions is shown in Listing 1. Shared memory creation involves two steps. First, we must create a *file mapping*, which is an abstract object that is of no direct use to us. To access the data in a mapped file, we must create a *view*. There is a logic behind this two-part approach. A file can be larger than 2Gb, and to be able to access such a huge file on Windows (where an address space for a program is limited to 2Gb), the operating system allows us to create a view that starts at an arbitrary position in the file.

Let's analyse the code now. File mapping is created with a call to the `CreateFileMapping` function. Its first parameter is the handle of the backing storage, which we will set to `INVALID_HANDLE_VALUE` as we want to use the page file for this purpose. We can then ignore the

➤ *Listing 1: Shared memory access via plain Windows API.*

```
program SharedMemoryTest1;
{$APPTYPE CONSOLE}
uses
  SysUtils, Windows;
var
  mapping      : THandle{CreateFileMapping};
  randomContent: integer;
  shmContent   : integer;
  view         : pointer{MapViewOfFile};
begin
  Randomize;
  Writeln('Program will now create a shared memory with '+
    ‘name "TDM_SHM_API"...');
  try
    mapping := CreateFileMapping(INVALID_HANDLE_VALUE, nil,
      PAGE_READWRITE, 0, 4, 'TDM_SHM_API');
    if mapping = 0 then
      RaiseLastWin32Error;
    try
      view :=
        MapViewOfFile(mapping, FILE_MAP_WRITE, 0, 0, 4);
      if view = nil then
        RaiseLastWin32Error;
      try
        randomContent := Random(1000);
        Writeln('Shared memory contents will be set to ',
          randomContent);
        integer(view^) := randomContent;
        Write('Veryfing ... ');
        shmContent := integer(view^);
        if shmContent = randomContent then
          Writeln(' OK')
        else
          raise Exception.CreateFmt(
            'Error! Shared memory containts value %d ‘+
            ‘instead of %d', [shmContent, randomContent]);
        Writeln('Start another instance of this program '+
         ‘to write a new value to the shared memory, ‘+
         ‘then press Enter.');
        Readln;
        Writeln('Shared memory now contains',
          integer(view^));
      finally
        if not UnmapViewOfFile(view) then
          RaiseLastWin32Error;
      end;
    finally
      if not CloseHandle(mapping) then
        RaiseLastWin32Error;
    end;
  except
    on E: Exception do
      Writeln(E.Message);
  end;
end.
```

```
program SharedMemoryTest2;
{$APPTYPE CONSOLE}
uses
  SysUtils, GpSharedMemory;
var
  randomContent: integer;
  sharedMemory : TGpSharedMemory;
  shmContent   : integer;
begin
  Randomize;
  Writeln('Program will now create a shared memory with '+
    'name "TDM_SHM_API"...');
  try
    sharedMemory :=
      TGpSharedMemory.Create('TDM_SHM_API', 4);
    try
      if sharedMemory.AcquireMemory(true, 1000) = nil then
        raise Exception.Create(
          'Failed to access shared memory.');
      try
        randomContent := Random(1000);
        Writeln('Shared memory contents will be set to ',
          randomContent);
        sharedMemory.Long[0] := randomContent;
        Write('Veriyfing ... ');
        shmContent := sharedMemory.Long[0];
        if shmContent = randomContent then
          Writeln(' OK')
        else
          raise Exception.CreateFmt('Error! Shared memory '+
            'contains value %d instead of %d',
            [shmContent, randomContent]);
      finally
        sharedMemory.ReleaseMemory;
      end;
      Writeln('Start another instance of this program to '+
        'write a new value to the shared memory, then '+
        'press Enter.');
      Readln;
      if sharedMemory.AcquireMemory(false, 1000) = nil then
        raise Exception.Create(
          'Failed to access shared memory.');
      try
        Writeln('Shared memory now contains ',
          sharedMemory.Long[0]);
      finally
        sharedMemory.ReleaseMemory;
      end;
    finally
      FreeAndNil(sharedMemory);
    end;
  except
    on E: Exception do
      Writeln(E.Message);
  end;
end.
```

➤ *Listing 2: Shared memory access via TGpSharedMemory class.*

security attributes (set to nil) and set access protection to PAGE_ READWRITE (we will both read and write). The next two parameters specify the maximum size of the file mapping: the first represents the higher 32 bits and the second is the lower 32 bits. As we will only share 4 bytes (an integer) we set the first parameter to 0 and the second to 4. The last parameter specifies the name of this file mapping. To be able to access one file mapping from more than one process, we must name it. When two processes create a file mapping with the same name, they will access the same underlying storage and therefore they will be sharing the data, which is exactly what we want to achieve.

Next, we will create a view of this mapping with a call to MapViewOfFile. The first parameter is the file mapping handle returned from the CreateFileMapping call. Next, we must specify the access protection (which may be more restrictive than the access protection specified in CreateFileMapping). FILE_MAP_ WRITE will give read and write access. Finally, we must specify the starting offset of the view inside the file mapping and the size of the view. To access every part of a potentially huge file, the starting offset is a 64-bit integer (specified as two 32-bit parts). The size is an ordinary 32-bit integer. To access all 4 bytes of our mapping, we must specify 0 for the offset and 4 for the size.

MapViewOfFile returns a pointer to the mapped memory. We can manipulate this pointer with normal memory access methods. For example, we can de-reference it and cast it into an integer. That is exactly how the demo accesses it.

The program will then write a random number into the shared memory and wait for the user to press Enter. At that moment, you can start another instance of the program, which will overwrite the shared memory with another random number. To verify that, press Enter in the first instance and you'll see the new number.

As you can see, we can make it work in less than 100 lines of code, but the solution is not a clean one. All that mumbo-jumbo with file mappings and views hides the purpose of the program. Accessing the data through pointers is not very safe. Worse, the program in Listing 1 has no access protection. It is entirely possible that two programs running at the same time would overwrite each other's data.

What we really need is a class that will simplify our work. In the first place, it must wrap the Win32 API calls into a more straightforward interface. It should also add access protection, simplify access to the shared memory data and, if possible, support resizing. All of this and more is included in TGpSharedMemory, which I will present on the next few pages. But first, let's see how the simple program from the beginning of the article would look using TGpSharedMemory (see Listing 2).

The first major change is in the uses section. The Windows unit is no longer needed. The next change is in the shared memory creation: the new code only creates one Delphi object. Also included is access protection: we use AcquireMemory to gain access to the shared memory and ReleaseMemory to release it. You can also see that an indexed property Long replaces the pointer wizardry from the first program. At first glance, TGpSharedMemory satisfies our requirements. But just to be sure, let's see how it is implemented.

A Delphi Way

As you can already guess, TGpSharedMemory uses the CreateFileMapping and MapViewOfFile APIs to do its dirty work. In addition, it will also use a mutex to guard a critical part of the shared memory creation and a Single Writer Multiple Readers guard to implement access protection.

The file mapping that we create in TGpSharedMemory will contain more than just the application data. The first 40 bytes will hold a shared memory header, whose declaration is shown in Listing 3. This header contains five important parameters that must be the same for all processes that share the shared memory block. Those five parameters are put between two guards containing magic value `$7E2D81FEF4E0BC22` (in case you wonder, this is a pseudo-random number without any hidden meaning). Guards are used to check for memory overrun problems.

To enable co-operation between different versions of `TGpSharedMemory` (although that is not an issue yet), the `gsmhVersion` field contains a version of the shared memory data structures. The current size of the shared memory (excluding the header, only application data is accounted for) is stored in field `gsmhSize` and the maximum size is stored in field `gsmhMaxSize`. The current number of allocated bytes for this shared memory block (see the discussion in the section *Resizing*) is stored in `gsmhAllocated`.

The last field, `gsmhModifiedCount`, starts its life at 0 and is incremented by 1 every time a writer releases shared memory (even if it didn't modify it). That allows readers to quickly decide if they should re-read shared memory data or not and is important when the processing of the contents of shared memory takes a long time (for example, when a text representation of an XML document is stored). `TGpSharedMemory` stores the last known value of this counter internally and, if it differs from the value stored in the header, the shared memory was modified. This can be checked for by reading the `Modified` property.

The constructor (see Listing 4) takes four parameters: the name of the shared memory (required), the initial size, the maximum size and a resource protection flag. The shared memory will be resizable (we'll come back to this) if the maximum size is non-zero. When shared memory is not resizable (the maximum size is zero), the initial size must be greater than zero. Finally, you can disable built-in access protection by setting the fourth parameter to `False`. The constructor verifies the correctness of the parameters, creates a SWMR guard and a mutex, and opens the shared memory.

The `OpenMemory` private method (Listing 4) is completely wrapped in the critical section (governed by the initialisation mutex created in the constructor). That is because we must do some non-atomic tricks to determine whether this was the first process to access the shared memory, and to execute some additional initialisation in that case.

Next, the code creates the file mapping. You'll note that it is always created with the `PAGE_READWRITE` protection flag regardless of how we will access the shared memory later. The true access protection will be set in the `MapViewOfFile` call. For now, you should ignore the `SEC_RESERVE` flag that is set if the shared memory is resizable. We will cover that later.

The execution then follows two different paths. If we are the first owner (the last error is `NO_ERROR`), then we must initialise the contents of the shared memory to zero and then create the internal header. If we are not the first owner (the last error is `ERROR_ALREADY_EXISTS`), we map the entire shared memory and check some parameters. All this initialisation and checking is done by the `MapView` function (see Listing 4). At the end, the mapped view is unmapped. The view is mapped only when the memory is acquired, to prevent an application from modifying the shared memory at the wrong time.

➤ *Listing 3: Shared memory header.*

```
TGpSharedMemoryHeader = packed record
  gsmhGuard1       : int64;    //Always containing CGpSharedMemorySignature.
  gsmhVersion      : cardinal; //Always 1. Used for possible future modifications
                               //of the header structure.
  gsmhSize         : cardinal; //Current size of the shared memory.
  gsmhMaxSize      : cardinal; //Maximum size of the shared memory.
  gsmhAllocated    : cardinal; //Size of the allocated memory.
  gsmhModifiedCount: int64;    //Counter showing how many times shared memory was
                               // modified.
  gsmhGuard2       : int64;    //Always containing CGpSharedMemorySignature.
end; { TGpSharedMemoryHeader }
```

Besides being used in the constructor, `MapView` is called from other parts of the code to map the view of the file, check the header, reallocate the memory (if the shared memory is resizable), etc. It starts by calculating the total size to be mapped (the requested size plus the size of the shared memory header). Next it maps this part of the shared memory with the desired access. If the memory is to be initialised, the shared memory header is filled with the values passed to the constructor. Otherwise, the header is checked for consistency (the signature stored in the guards must be correct, the version must be 1), the `Size` and `MaxSize` properties are compared to the header values and possible errors are reported. If the shared memory is resizable, the `Size` property is updated from the header and the view is mapped again with the new size. The last part (the part that calls `VirtualAlloc`) handles the virtual memory resizing (and will, as you may have already guessed, be discussed later).

That pretty much covers the shared memory creation. The destruction part is much easier. The destructor (see Listing 4) releases the memory if it is currently acquired (unmapping the view in progress), closes the file mapping, destroys the initialising mutex and the SWMR guard.

Access Protection

Every experienced programmer can tell you that it is really important to synchronise access to shared resources. It is OK for two processes to read the same data, but it is usually a very bad thing to allow one reader and one writer (and it is an even worse thing to allow two simultaneous writers). To prevent such incidents, we would traditionally use some separate mechanism of access control. Typically, a mutex is used to wrap shared memory accesses into a critical section.

`TGpSharedMemory` uses a slightly more sophisticated approach allowing multiple readers to access shared memory at the same time. The synchronisation primitive used, the Single Writer Multiple Readers guard, was presented in Issue 86. In short, it allows the shared memory (or any other shared resource) to be accessed either by a single writer or by multiple readers.

To access shared memory, a program calls `AcquireMemory` (Listing 5), with two parameters: the type of access required (read or write) and the timeout value (in milliseconds, `INFINITE` is supported). If the function can grant the required access in the allotted time, it will return a raw pointer to the start of the shared memory block. Otherwise, it will return `nil`.

If we skip the error handling, `AcquireMemory` first accesses the internal `TGpSWMR` object and requires write or read access. If that succeeds, it calls the `MapView` workhorse (which we have already described) to map a view of the shared memory into the process space.

The process may disable internal access protection handling by

➤ *Listing 4: Shared memory creation and destruction.*

```
constructor TGpSharedMemory.Create(objectName: string; size,
  maxSize: cardinal; resourceProtection: boolean);
begin
  inherited Create;
  if objectName = '' then
    raise EGpSharedMemory.Create(sNameRequired);
  SetName(objectName);
  inherited SetSize(size);
  if (size = 0) and (maxSize = 0) then
    raise EGpSharedMemory.CreateFmt(
      sInvalidSharedMemorySize, [Name, Size]);
  SetMaxSize(maxSize);
  if (maxSize > 0) and (size > maxSize) then
    raise EGpSharedMemory.CreateFmt(
      sSizeMustBeSmallerThanMaxSize, [Name, size, maxSize]);
  if resourceProtection then
    gsmSynchronizer := TGpSWMR.Create(Name+'$SWMR');
  gsmInitializer := CreateMutex(nil, false,
    PChar(Name+'$MTX'));
  if gsmInitializer = 0 then
    RaiseLastWin32Error;
  SetCreated(OpenMemory);
end; { TGpSharedMemory.Create }
function TGpSharedMemory.OpenMemory: boolean;
var
  fPtr           : pointer;
  protectionFlags: DWORD;
begin
  Result := false; // to keep Delphi happy
  if WaitForSingleObject(gsmInitializer,
    CInitializationTimeout*1000) <> WAIT_OBJECT_0 then
    raise EGpSharedMemory.CreateFmt(sInitializationTimeout,
      [Name, SysErrorMessage(GetLastError)])
  else begin
    try
      protectionFlags := PAGE_READWRITE;
      if IsResizable then
        protectionFlags := protectionFlags OR SEC_RESERVE;
      gsmFileMapping := CreateFileMapping(
        INVALID_HANDLE_VALUE, nil, protectionFlags, 0,
        SizeOf(TGpSharedMemoryHeader)+GetUpperSize,
        PChar(Name));
      if gsmFileMapping = 0 then
        RaiseLastWin32Error
      else begin
        if GetLastError = NO_ERROR then begin
          // first owner, initialize to 0 and write header
          fPtr := MapView(gsmFileMapping, FILE_MAP_WRITE,
            Size, false, true);
          UnmapView(fPtr);
          Result := true;
        end else if GetLastError = ERROR_ALREADY_EXISTS
          then begin
          // not first owner, check size if not resizable
          fPtr := MapView(gsmFileMapping, FILE_MAP_READ);
          UnmapView(fPtr);
          Result := false;
        end elee
          // not (GetLastError in
          // [NO_ERROR,ERROR_ALREADY_EXISTS])
          RaiseLastWin32Error;
      end; //else gsmFileMapping = 0
    finally
      ReleaseMutex(gsmInitializer);
    end;
  end; //else WaitForSingleObject()
end; { TGpSharedMemory.OpenMemory }
function TGpSharedMemory.MapView(mappingObject: THandle;
  desiredAccess: DWORD; mappingSize: DWORD; getFromHeader:
  boolean; initialize: boolean): pointer;
var
  allocSize: DWORD;
  header   : PGpSharedMemoryHeader;
  totalSize: cardinal;
begin
  if initialize and getFromHeader then
    raise Exception.Create('GpSharedMemory: Internal ‘+
      ‘error. Initialize and getFromHeader are both True ‘+
      ‘in MapView.'); //DNT
  totalSize := mappingSize + SizeOf(TGpSharedMemoryHeader);
  Result := MapViewOfFile(mappingObject, desiredAccess, 0,
    0, totalSize);
  if not assigned(Result) then
    RaiseLastWin32Error;
  header := PGpSharedMemoryHeader(Result);
  if initialize then begin
    if not IsResizable then
      allocSize := 0
    else begin
      allocSize := totalSize;
      if VirtualAlloc(header, allocSize, MEM_COMMIT,
        PAGE_READWRITE) = nil then
        RaiseLastWin32Error;
    end;
    header^.gsmhGuard1    := CGpSharedMemorySignature;
    header^.gsmhVersion   := 1;
    header^.gsmhSize      := mappingSize;
    header^.gsmhMaxSize   := MaxSize;
    header^.gsmhAllocated := RoundToNextPage(allocSize);
    header^.gsmhGuard2    := CGpSharedMemorySignature;
  end else begin
    if (header^.gsmhGuard1 <> CGpSharedMemorySignature) or
       (header^.gsmhGuard2 <> CGpSharedMemorySignature) then
      raise EGpSharedMemory.CreateFmt(sMemoryBlockCorrupted,
        [Name])
    else if header^.gsmhVersion <> 1 then
      raise EGpSharedMemory.CreateFmt(sInvalidHeaderVersion,
       [Name, header^.gsmhVersion]);
    if not getFromHeader then
      header^.gsmhSize := mappingSize
    else begin
      if header^.gsmhMaxSize <> MaxSize then
        raise EGpSharedMemory.CreateFmt(
          sSMAlreadyExistsMaxSizeDiffers, [Name,
          header^.gsmhMaxSize, MaxSize])
      else if (MaxSize = 0) and (header^.gsmhSize <> Size)
        then
        raise EGpSharedMemory.CreateFmt(
          sSMAlreadyExistsSizeDiffers, [Name,
          header^.gsmhSize, Size]);
      inherited SetSize(header^.gsmhSize);
      inherited SetMaxSize(header^.gsmhMaxSize);
      if mappingSize <> Size then begin
        UnmapView(Result);
        mappingSize := Size;
        totalSize := mappingSize +
          SizeOf(TGpSharedMemoryHeader);
        Result := MapViewOfFile(mappingObject,
          desiredAccess, 0, 0, totalSize);
        if not assigned(Result) then
          RaiseLastWin32Error;
        header := PGpSharedMemoryHeader(Result);
      end;
    end;
  end;
  if IsResizable then begin
    if totalSize > header^.gsmhAllocated then begin
      if VirtualAlloc(Ofs(header, header^.gsmhAllocated),
           totalSize - header^.gsmhAllocated,
           MEM_COMMIT, PAGE_READWRITE) = nil then
        RaiseLastWin32Error;
      header^.gsmhAllocated := RoundToNextPage(totalSize);
    end;
  end;
  SetDataPointer(Ofs(Result,
    SizeOf(TGpSharedMemoryHeader)));
end; { TGpSharedMemory.MapView }
destructor TGpSharedMemory.Destroy;
begin
  if Acquired then
    ReleaseMemory;
  if gsmFileMapping <> 0 then begin
    CloseHandle(gsmFileMapping);
    gsmFileMapping := 0;
  end;
  if gsmInitializer <> 0 then begin
    CloseHandle(gsmInitializer);
    gsmInitializer := 0;
  end;
  FreeAndNil(gsmSynchronizer);
  inherited;
end; { TGpSharedMemory.Destroy }
```

```
function TGpSharedMemory.AcquireMemory(forWriting: boolean;
  timeout: DWORD): pointer;
var
  gotAccess: boolean;
begin
  if gsmFileMapping = 0 then
    raise EGpSharedMemory.CreateFmt(
      sTryingToAcquireNoninitialized, [Name])
  else begin
     if Acquired then
      raise EGpSharedMemory.CreateFmt(sMissingReleaseMemory,
        [Name]);
    if not assigned(gsmSynchronizer) then
      gotAccess := true
    else if forWriting then
      gotAccess := gsmSynchronizer.WaitToWrite(timeout)
    else
      gotAccess := gsmSynchronizer.WaitToRead(timeout);
    if not gotAccess then
      Result := nil
    else begin
      SetIsWriting(forWriting);
      if forWriting then
        gsmDesiredAccess := FILE_MAP_WRITE
      else
        gsmDesiredAccess := FILE_MAP_READ;
      grmFileView := MapView(gsmFileMapping,
        gsmDesiredAccess);
      Result := DataPointer;
      if not assigned(Result) then
        RaiseLastWin32Error;
    end;
  end;
end; { TGpSharedMemory.AcquireMemory }
procedure TGpSharedMemory.ReleaseMemory;
begin
  if not Acquired then
    raise EGpSharedMemory.CreateFmt(sNotAcquired, [Name])
  else begin
    if IsWriting and HaveStream then begin
      ResizeMemory(CopyStream.Size);
      CopyStream.Position := 0;
      CopyStream.Read(DataPointer^,Size);
      FreeStream;
    end;
    if IsWriting then
      Inc(PGpSharedMemoryHeader(
        grmFileView)^.gsmhModifiedCount);
    gsmModifiedCount := PGpSharedMemoryHeader(
      grmFileView)^.gsmhModifiedCount;
    try
      UnmapView(grmFileView);
    finally
      if assigned(gsmSynchronizer) then
      gsmSynchronizer.Done;
    end;
  end;
  inherited;
end; { TGpSharedMemory.ReleaseMemory }
```

➤ *Listing 5: Acquiring and releasing shared memory.*

specifying `resourceProtection := false` in the `TGpSharedMemory` constructor. In that case, the internal `TGpSWMR` object is not allocated at all. Nevertheless, the program must still call `AcquireMemory` to access the shared memory.

When a program doesn't need shared memory any more, it must release its hold on it by calling the `ReleaseMemory` method (see Listing 5). This first updates the memory contents if the stream interface is used (we'll discuss this in the section *Stream Interface*), updates the modification counter, unmaps the view and releases the internal `TGpSWMR` object (if access protection is active).

If your code takes a long time to read the data in shared memory, you may be concerned that reading will block out writers for too long. In that case, you can create a read-only snapshot of the shared memory contents, release the memory, and continue to work on the snapshot.

The interesting thing about the snapshot object `TGpSharedSnapshot` (created with the function `MakeSnapshot`) is that it implements almost the same interface as `TGpSharedMemory`. In fact, both classes are descendants of `TGpBaseSharedMemory`, which defines this interface. With some precautions, you can easily write code that doesn't care if it is working with real shared memory or a shared snapshot.

```
function TGpBaseSharedMemory.GetAsString: string;
begin
  if not Acquired then
    raise EGpSharedMemory.CreateFmt(sNotAcquired, [Name]);
  if Size < 4 then
    raise EGpSharedMemory.CreateFmt(sNotAString, [Name]);
  SetLength(Result, Long[0]);
  if Length(Result) > 0 then
    Move(Ofs(DataPointer, 4)^, Result[1], Length(Result));
end; { TGpBaseSharedMemory.GetAsString }
procedure TGpBaseSharedMemory.SetAsString(const Value: string);
begin
  if not Acquired then
    raise EGpSharedMemory.CreateFmt(sNotAcquired, [Name]);
  if not IsWriting then
    raise EGpSharedMemory.CreateFmt(sNotAcquiredForWriting, [Name]);
  if cardinal(Length(Value)+4) > GetUpperSize then
    raise EGpSharedMemory.CreateFmt(sStringIsTooLong, [Name, GetUpperSize]);
  if IsResizable then
    ResizeMemory(Length(Value)+4);
  Long[0] := Length(Value);
  if Length(Value) > 0 then
    Move(Value[1], Ofs(DataPointer,4)^, Length(Value));
end; { TGpBaseSharedMemory.SetAsString }
```

➤ *Listing 6: Treating shared memory as a string.*

Text Access

You already know that `TGpSharedMemory` allows access to a shared memory block through a raw pointer returned from the `AcquireMemory` function. That is not all the class has to offer, though.

The easiest way to access the shared memory is with the `AsString` property. It allows you to treat the shared memory as a string. The first four bytes of the shared memory contain the length of the string and the rest contains the string itself.

The getter (`GetAsString`) and setter (`SetAsString`) for this property are fairly trivial (see Listing 6). The getter allocates a string of the required length and moves the data from the shared memory into the string, while the setter resizes the shared memory to fit the string (if the shared memory is resizable) and stores the string length and contents into it. Both methods use the `Long` property to access the first 4 bytes of the shared memory. We met it in the second example (see Listing 2) but then we silently ignored it. The time has finally come: meet `Long` and its friends, the indexed accessors.

Indexed Access

`TGpSharedMemory` implements four properties that allow you to treat the shared memory as an array of numbers. `ByteIdx` 'sees' the shared memory as a sequence of bytes, `WordIdx` thinks that the shared memory is filled with two-byte words, LongIdx accesses four-byte integers, and HugeIdx works with eight-byte numbers.

All the properties are indexed by a number and all start counting from zero. That is, ByteIdx[0] returns the first byte of the shared memory, WordIdx[0] returns the first two bytes, and so on.

There is no heavy magic behind these properties. They all have pretty dull getters and setters, that essentially call the methods GetData and SetData, shown in Listing 7 together with accessors for the property WordIdx.

Everything should be pretty obvious except for two previously unmentioned functions. CheckDMA makes sure that the stream interface (see next section) is not active, and CheckBoundaries verifies that the data required is completely inside the shared memory area.

If your shared memory layout is not that regular, you can maybe use four similar properties: Byte, Word, Long, Huge. Like their -Idx counterparts they access one- to eight-byte numbers. The difference lies in the index. These properties treat the shared memory as a formless memory blob and the index represents the offset of the data inside this blob. In other words, Byte[0] returns the first byte of the shared memory, Word[1] returns the word represented by bytes 1 and 2 (compare this to WordIdx[1], which returns second word, that is bytes 2 and 3), Long[2] returns the integer represented by bytes 2 to 5, and so on.

If that is still not formless enough for your taste, you obviously won't mind accessing shared memory as a stream. As you may have guessed, TGpSharedMemory has a stream interface built in.

```
function TGpBaseSharedMemory.GetWordIdx(idx: integer): word;
begin
  CheckDMA;
  GetData(idx*SizeOf(Result), SizeOf(Result), Result);
end; { TGpBaseSharedMemory.GetWordIdx }
procedure TGpBaseSharedMemory.SetWordIdx(idx: integer; Value: word);
begin
  CheckDMA;
  SetData(idx*SizeOf(Value), SizeOf(Value), Value);
end; { TGpBaseSharedMemory.SetWordIdx }
procedure TGpSharedMemory.GetData(offset, size: cardinal; out buffer);
begin
  if not Acquired then
    raise EGpSharedMemory.CreateFmt(sNotAcquired, [Name]);
  CheckBoundaries(offset, size);
  Move(Ofs(DataPointer,offset)^, buffer, size);
end; { TGpSharedMemory.GetData }
procedure TGpSharedMemory.SetData(offset, size: cardinal; var buffer);
begin
  if not Acquired then
    raise EGpSharedMemory.CreateFmt(sNotAcquired, [Name]);
  if not IsWriting then
    raise EGpSharedMemory.CreateFmt(sNotAcquiredForWriting, [Name]);
  CheckBoundaries(offset, size);
  Move(buffer, Ofs(DataPointer,offset)^, size);
end; { TGpSharedMemory.SetData }
```

➤ *Listing 7: Indexed access to the shared memory.*

Stream Interface

Sometimes it is useful to treat shared memory as a stream that can be passed to an existing piece of code. For example, it is quite simple to store an XML document in the shared memory in this way. The stream interface is most powerful when it is used with resizable shared memory, but you can use it with fixed-size shared memory too.

When you retrieve the stream interface to the shared memory through the property AsStream, the shared memory behavior changes. Text access stops working, and so does indexed access. Access through the raw pointer (the DataPointer property) is still available, but is not really recommended, as data stored in the shared memory block may not be in sync with the streamed version.

The stream interface getter (GetAsStream, see Listing 8) creates an instance of the TGpSharedStream class if one does not already exist. Every instance of TGpSharedMemory has at most one instance of TGpSharedStream associated with it. That instance (we'll call it the *shared stream*) remaps TStream-type access (Read, Write, Seek...) into relevant shared memory calls.

When writing into the shared stream, a few different scenarios can occur. The most simple version is that the data being written fits into the shared memory. In that case, it is copied into the right place (using raw pointer access). If there is not enough space for all the data and the shared memory is not resizable, as much data as possible is stored. The Write function returns the number of bytes written (see Listing 9).

Interesting things start to happen if the shared memory is resizable. In that case, the internal TMemoryStream instance is created and all the data from the shared memory is copied into it. All the subsequent operations on shared stream work on that copy. Only when the shared memory is released (see ReleaseMemory in Listing 5: the block starting with if IsWriting and HaveStream), it is resized to accommodate the new data and the content of the stream is copied back into the shared memory.

Reading from the stream (see Listing 9) is simpler than writing. If the internal TMemoryStream is created, the data will be taken from it, otherwise the data from the shared memory will be used.

➤ *Listing 8: Retrieving the stream interface.*

```
function TGpBaseSharedMemory.GetAsStream: TGpSharedStream;
begin
  if not Acquired then
    raise EGpSharedMemory.CreateFmt(sNotAcquired, [Name]);
  if not HaveStream then
    gbsmStream := TGpSharedStream.Create(Self);
  Result := gbsmStream;
end; { TGpBaseSharedMemory.GetAsStream }
```

Resizing

So far, we have achieved all our objectives except resizing, and that's the tough one. Windows doesn't really offer much help in that direction. The size of a shared memory block must be known when it is created and it cannot be resized after that.

The only thing we can do is to conserve the memory. We can create a shared memory block with a very large size that doesn't use any memory. Later, we can allocate just the memory that we need. This is possible because memory allocation in Windows is a two-stage process. In the first stage, the operating system reserves a part of a process's virtual address space without allocating any physical storage (the process is called *reserving*). In the second stage, the OS allocates physical memory to back that virtual address space (*committing*). Usually, both stages happen inside one memory allocation call, but we are free to request only the first or only the second stage (for more information, see the Help on the `VirtuallAlloc` API function).

This trick is complicated for two reasons. On the NT platform, file mapping memory can only grow. It is possible to allocate physical memory to match the pre-allocated virtual memory but it is impossible to release it. Memory use of the resizable shared memory can therefore only grow (until all processes release the shared memory and the OS destroys it, of course). It is possible to shrink physical memory associated with the file mapping on Windows 9x, but for the sake of simplicity I have decided to ignore that fact. That is why `MapView`, which implements physical memory allocation, only calls `VirtualAlloc` and never calls `VirtualFree`.

The other problem appears on the Windows 9x platform. Here, the OS wants to reserve enough space in the page file for all the memory we *might* want to use. If you create a shared memory block with a very large maximum size, you will most probably run out of space in your page file, or your disk space, whichever comes first. There is no good solution for this problem. Either design your shared memory use with great care, or focus on the NT platform.

The last thing you may want to know is that all this only works with file mappings backed by the page file. You can't do that trick if `CreateFileMapping` is working on a real file. But, as we have designed shared memory to work with page file, this isn't really an issue.

Given all that, the resizable support is just a matter of correct implementation. The first part, reserving, is done in the `OpenMemory` method (see Listing 4) where `SEC_RESERVE` flag is used in the call to `CreateFileMapping`. This affects the `MapView` method (also shown in Listing 4) where `MapViewOfFile` is called. When `SEC_RESERVE` is set, `MapViewOfFile` returns a pointer as usual, but at that time the pointer is not pointing to the physical memory. It is pointing to virtual memory that is not backed by the physical memory. To allocate the required physical memory we must call `VirtualAlloc` with the `MEM_COMMIT` flag. When our data grows, more physical memory may be required and `VirtualAlloc` must be called again. All that is complicated by the fact that `VirtualAlloc` commits memory in chunks: the size that we pass to it will be rounded up to the next page boundary. On the Intel architecture, and that is the only one that interests us, a page size is 4Kb, meaning that every memory allocation is always rounded up to the next 4Kb. All that trickery is handled (hopefully in a correct manner) in the `MapView` method.

Component Wrapper

Well, this is it: a shared memory class in all its details. This month's source, however, hides some other treasures. There is, for example, a `GpSharedMemoryComp` unit containing a component wrapper for the `TGpSharedMemory` class.

The shared memory wrapper `TGpSharedMemoryComp` exposes most of the shared memory functionality through a number of public and published properties (see Listing 10).

To activate the shared memory, you have to set its name (using the property `SharedMemoryName`), size (use `InitialSize` for fixed-size memory or `MaxSize` for resizable memory), and flip the `Active` property to `True`. That will create an internal `TGpSharedMemory` instance

➤ *Listing 9: Reading and writing a shared memory stream.*

```
procedure TGpSharedStream.CopyOnWrite;
var memSize: cardinal;
begin
  //Must get shared memory size before stream is created,
  //because after that TGpResizableSharedMemory returns
  //stream size as its size.
  memSize := Memory.Size;
  ssCopyStream := TMemoryStream.Create;
  CopyStream.Write(Memory.DataPointer^, memSize);
  CopyStream.Position := MemoryPos;
end; { TGpSharedStream.CopyOnWrite }
function TGpSharedStream.Read(var buffer; count: integer):
  longint;
var
  remaining: integer;
begin
  if UseStream then
    Result := CopyStream.Read(buffer, count)
  else if MemoryPos < Memory.Size then begin
    remaining := Memory.Size-MemoryPos;
    if remaining > count then
      remaining := count;
    Move(CurrentData^, buffer, remaining);
    MemoryPos := MemoryPos + cardinal(remaining);
    Result := remaining;
  end else
    Result := 0;
end; { TGpSharedStream.Read }
function TGpSharedStream.Write(const buffer; count:
  integer): longint;
var remaining: integer;
begin
  if not Memory.IsWriting then
    raise EGpSharedMemory.CreateFmt(sNotAcquiredForWriting,
    [Memory.Name]);
  ssModified := true;
  if UseStream then
    Result := CopyStream.Write(buffer, count)
  else begin
    remaining := Memory.Size-MemoryPos-1;
    if remaining > count then
      remaining := count;
    if (remaining < count) and Memory.SupportsResize then
      begin
      CopyOnWrite;
      Result := Write(buffer, count);
    end else begin
      Move(buffer, CurrentData^, remaining);
      SetMemoryPos(int64(MemoryPos) + int64(remaining));
      Result := remaining;
    end;
  end;
end; { TGpSharedStream.Write }
```

```
TGpSharedMemoryComp = class(TComponent)
public
  constructor Create(AOwner: TComponent); override;
  destructor  Destroy; override;
  function  AcquireMemory(forWriting: boolean; timeout: DWORD): pointer;
  procedure ReleaseMemory;
  property AsStream: TGpSharedStream read GetAsStream;
  property Byte[byteOffset: integer]: byte read GetByte write SetByte;
  property ByteIdx[idx: integer]: byte read GetByteIdx write SetByteIdx;
  property DataPointer: pointer read GetDataPointer;
  property Huge[byteOffset: integer]: int64 read GetHuge write SetHuge;
  property HugeIdx[idx: integer]: int64 read GetHugeIdx write SetHugeIdx;
  property Long[byteOffset: integer]: longword read GetLong write SetLong;
  property LongIdx[idx: integer]: longword read GetLongIdx write SetLongIdx;
  property Word[byteOffset: integer]: word read GetWord write SetWord;
  property WordIdx[idx: integer]: word read GetWordIdx write SetWordIdx;
published
  property Access: TGpSharedMemoryAccess read FAccess write SetAccess;
  property Acquired: boolean read GetAcquired write IgnoreSetAcquired;
  property AsString: string read GetAsString write SetAsString;
  property InitialSize: cardinal read FInitialSize write SetInitialSize;
  property IsResizable: boolean read GetIsResizable write IgnoreSetIsResizable;
  property MaxSize: cardinal read FMaxSize write SetMaxSize;
  property Modified: boolean read GetModified write IgnoreSetModified;
  property SharedMemoryName: string read FSharedMemoryName write
SetSharedMemoryName;
  property Size: cardinal read GetSize write SetSize;
  property Timeout: DWORD read FTimeout write FTimeout;
  property WasCreated: boolean read GetWasCreated write IgnoreSetWasCreated;
  property Active: boolean read FActive write SetActive;
end; { TGpSharedMemoryComp }
```

➤ *Listing 10: TGpSharedMemory component wrapper.*

(and setting `Active` to `False` will destroy it).

The `AcquireMemory` and `ReleaseMemory` methods are replaced with the `Access` property, which has three possible states: `accNone`, `accRead` and `accWrite`. Setting it to the `accNone` state releases memory, while using any of the other two states calls `AcquireMemory` with the timeout equal to the value of the `Timeout` property. If the required state cannot be acquired, `Access` will revert to `accNone`. When memory is required, the read-only property `Acquired` becomes `True`.

To modify the shared memory use the published property `AsString`, which gives access to the text interface, the public property `AsStream`, or the public index accessors (`Byte`, `ByteIdx`, etc).

The job on the shared memory class is done and we can focus on bigger issues. Next time, we'll discuss how to use shared memory in client-server solution containing multiple data producers and one data processor. Believe me, it is not as trivial as it may seem.

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

All code in this article is freeware and may be freely reused in your own applications.