Programming Games for Series 60

EXCERPTS FROM THE MASTER OF SCIENCE THESIS
”SERIES 60 AND SYMBIAN OS BASED SMART PHONE AS A MULTITERMINAL GAME PLATFORM”
BY JUUSO KANNER
TAMPERE UNIVERSITY OF TECHNOLOGY 2002

Version 1.0
August 5, 2003

Contents

Legal Notice

Copyright © 2003 Nokia Corporation. All rights reserved.

Copyright © of the original document Juuso Kanner 2002. All rights reserved.

Reproduction, transfer, distribution, or storage of part or all of the contents in this document in any form without the prior written permission of Nokia is prohibited.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Nokia operates a policy of continuous development. Nokia reserves the right to make changes and improvements to any of the products described in this document without prior notice.

Under no circumstances shall Nokia be responsible for any loss of data or income or any special, incidental, consequential, or indirect damages howsoever caused.

The contents of this document are provided "as is." Except as required by applicable law, no warranties of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, are made in relation to the accuracy, reliability, or contents of this document. Nokia reserves the right to revise this document or withdraw it at any time without prior notice.

1. Introduction

This document is based on the Master of Science thesis, ”Series 60 and Symbian OS Based Smart Phone as a Multiterminal Game Platform,” by Juuso Kanner, Tampere University of Technology, 2002. It contains only excerpts of the thesis that are relevant for the development of advanced games on current Series 60 terminals.

The following chapters have been removed from the original document:

1. Introduction

2. Symbian Operating System Based Smart Phones
(except 2.3 Symbian Operation System)

6. Implementation of a Multiterminal Game for Series 60

7. Conclusion

The references have been updated. Any subsequent additions by Nokia are marked.

Note: This document discusses Symbian OS GT 6.1 as the basis of Series 60 Platform. This is true for Series 60 Platform v1.x. Series 60 Platform v2.0 is based on Symbian OS GT 7.0s. This document is primarily applicable to Series 60 Platform v1.x and may be partially incompatible with Series 60 Platform v2.0.

2. Symbian Operating System

Symbian operating system is the common core of application programming interfaces (APIs) technology that is shared by all Symbian OS phones. The core is named as generic technology (GT) and it is divided into different releases. *The GT includes a multi-tasking kernel, middleware for communications, data management and graphics, the lower levels of the GUI framework, and application engines. [Sy02].*

Small hand-held devices, such as smart phones, are usually very resource-constrained devices. Size of the device and manufacturing costs constrict the memory available, processing speed and battery-life. Despite the scarce resources, the device needs to remain stable for a long period of time, even for months. In the case of an out-of-resources error, it is important for the system to return to the former state that was stable, without losing any vital data. This makes it important for the system and applications to catch and handle every run-time error properly.

Errors arising from out-of-resources, like all run-time errors are called exceptions. In standard C++ these exceptions are handled with a try-catch-and-throw mechanism, but because of its negative impact on code size, Symbian OS provides its own mechanism called trap harness. Another reason for Symbian to develop their own exception handler was that, at the time that Symbian OS was originally developed, the try-catch-and-throw mechanism was not a part of the C++ standard. The concept of the trap-harness is to encapsulate functions that may raise an exception with a `TRAP` macro. The macro can be used to trap multiple functions, and the functions may be nested. In the case of an exception, the execution of the function that caused it, is terminated by calling `User::Leave` function, which corresponds to throw in standard C++ exception handling. This is called a leave, and it will return the program execution to the closest `TRAP` macro, where suitable recovery actions can be performed. Symbian OS also provides a tool for cleanup in case of an exception. A cleanupstack is used to refer to objects that are only referred to by an automatic variable, and which need to be deallocated if a leave occurs. The `TRAP` macro will destroy the memory allocated by the automatic variables in the cleanupstack.

Symbian OS provides a system for non-preemptive multitasking within a single thread. The system, including active objects and an active scheduler, were designed to cut down run-time costs and synchronization problems encountered with preemptively scheduled threads. Every application in Symbian OS consists of an active scheduler and one or more active objects. The scheduler encapsulates a wait loop needed for asynchronous services and schedules active objects according to their priorities. The active objects encapsulate the actual ansynchronous services. More information about the non-preemptive multitasking infrastructure of the Symbian OS can be found from reference [Ta00].

3. Series 60 Based Smart Phone as a Device for Games

This chapter takes a deeper look into Series 60 and Symbian OS, and describes their characteristics as a platform for games. In addition, the requirements and restrictions that are set by smart phones are discussed. The chapter is mainly based on references: [Ta00], [Sy99] and [No01a].

3.1 Requirements

Unlike to many other devices that are used for gaming, smart phones need to able to inform a user about different system events while a game, or any other application is running. Applications need to take into account possible interruptions for example due to an incoming call or message, and they need to act accordingly. The applications should also not reserve or consume device resources, like memory or battery-life excessively.

Most of the system messages are indicated to a user with a system owned dialog, called a global note. The dialogs have a higher window priority than applications and thus they appear in the front of the applications. One exception in the system events is an incoming call, which causes telephony application to become the frontmost application, leaving the interrupted application background. However, all system side events have a common characteristic, which can be caught by an application. When a system event occurs, the frontmost application loses focus. This causes application user interface class' (`CAknAppUI`) `HandleForegroundEventL` method to be called. By overriding the method, applications can perform needed actions and, for example, pause the ongoing game.

Applications need to pay attention to battery consumption. When a phone is unused for a predefined amount of time it goes to a sleep mode to minimize power consumption. That can not be done, if an application continues doing background processing and for example polls a variable in a loop. All polling should be done in blocking loops, and all timers should be stopped when a game is paused. In case a timer is needed to maintain a connection to another terminal, the timer's frequency should be lowered to minimum. Applications can also get events from a system side timer when there has been no user activity for a predefined interval of time. This is done using the `RTimer::Inactivity` method that can be found from e32std.h header file. In battery-powered devices the software needs to be prepared for a sudden loss of battery power. The battery may fail or a user may remove it from the device. This should be noted if important user data is edited. The data should be saved at intervals, and restored after a reboot. In addition, applications should be prepared for corrupted data, and recover from the situations of that kind safely.

3.2 Restrictions

In addition to restricted memory size, smart phones have several other restrictions when compared to PCs. First of all, smart phones do not have as efficient processors as PCs do. Math processors are also very rare in smart phones and hence the time critical calculations should be implemented using integers. Symbian OS has also some constraints as a platform for games. For instance, writable static data, which is often used in games to optimise access to widely used data, is not supported by the platform.

Smart phones have also relatively limited hardware. Displays have limited resolution, size and colour depth. Keypads have a limited number of keys and the layout of the keys may be disadvantageous for playing games. The layout may also vary between different hardware solutions, and hence games should provide a possibility for users to redefine keys. Smart phones do not either have adequate ports to support various game controllers, like wheels and joysticks, which are familiar from the PC environment. All these restrictions determine what kinds of games can be implemented and ported to a smart phone, without loosing their playability. In the long run, however, technologies used in smart phones will elaborate, and new features and solutions will be introduced.

3.3 Memory

In memory-constrained devices, the memory management is in a very important position. This concerns both the run time memory usage and the eventual compiled code size. Most of the Symbian OS and the Series 60 based devices have only 8 MB of RAM, or less. In addition to RAM, the devices have ROM for preinstalled software and a user data area, which is used for installed applications and the system's writeable and persistent data files. In addition, a portable memory card, like a compact flash (CF) card or a multimedia card (MMC), may be supported depending on the hardware.

The most important rule for RAM usage is that all allocated memory should be deallocated at as early a stage as possible. The Symbian OS emulator provides a macro for memory checking, which is supplemented by default to all applications having a graphical UI (GUI). The macro will panic an application if it does not deallocate its memory and thus exposes all memory leaks at an early stage of an application development. On a target hardware, OS's kernel keeps track of every thread's memory and deallocates it automatically when a thread exits. This ensures that all memory is deallocated when an application exits. A problem may occur in applications or servers that are running for a long period of time. If they do not release unneeded resources after they have finished using them, a significant amount of resources may be reserved in the system and from other applications.

When implementing an application, the usage of a stack memory is worth noting. In Symbian OS each thread has its own memory stack, which cannot grow after the thread has been launched. The default stack size for an application in Series 60 is only 20 kB, so it should be used with great caution. There is also divergence between the emulator environment and the target hardware in stack space available. The stack size in emulator is not as constrained as on hardware because the Windows' own stack is used instead. This is why all software should be tested on hardware at as early a stage as possible, and with stack variables having their maximum sizes. Most of the stack overflows are caused by the use of stack descriptors. This can be avoided by allocating descriptors from the heap and by using automatic objects only for very short strings. Also the usage of recursion can be a very stack consuming. If recursive programming is necessary, the sizes of the passed parameters and the local automatic variables inside the recursive part should be minimized.

To minimize the size of the compiled code, the following guidelines should be followed:

- do not export methods unless it is necessary,
- do not create unnecessary virtual methods,
- do not use TRAPs excessively,
- avoid duplicate code,

- o find decomposable functions, and
- o use common controls and components

To enable accessing of a function or data from outside of a DLL, exported methods are listed in a DLL export table. Although in Symbian OS, the methods are exported by ordinal and not by name, all unnecessarily exported methods grow the size of the export table vainly. This is why methods should be exported only if they are designed to be used outside of the library they were introduced in. The same applies to virtual methods which are listed in a virtual function table of a DLL.

The usage of the TRAP macros should be carefully designed. They are not meant to be used excessively because of their negative impact on the size of the compiled code. Most often the TRAPs provided by the Symbian OS's application framework, are enough for application developers and they do not need to code their own TRAPs.

The last three items in the list are very common ways to minimize the code size for all platforms, and do not need to be discussed in more details.

Due to the graphical nature of games, bitmaps often form a large portion of their memory consumption. This applies for both the RAM and the user data area consumption. The most effective way to contribute to the consumption, without decreasing the number of bitmaps is to reduce their colour depths. Symbian OS supports 24 bit bitmaps which equals to 16777216 colours, but the actual maximum number of colour is constricted by the target hardware. This is why all bitmaps should not be converted to higher than the maximum colour depth set by the hardware. Small, low-detailed bitmaps, which do not need that many colours, should be converted to lower colour depth than the maximum colour depth referred to above. For example 8 bit colours are suitable for most of the sprites. All masks should be converted to 1 bit bitmaps.

3.4 Timers

Timing is essential for most of the games. Timing services, provided by the system side, are used for different purposes. In more complex games the game world and graphics are updated, and user input is read dozens of times in a second. In more straightforward games, timers are used to handle players' turns or to evaluate a player's success in resolving a given problem and so on. Most of the games need some kind of timing support from the system.

One of the most deficient services for game developers in Symbian OS are timing services. The OS does not support low level timer interrupts, and it only provides a kernel side timer which has the maximum frequency of 64 Hz. The same tick rate is also used for round-robin scheduling of threads. In the emulator environment the maximum tick rate is 10 Hz which makes testing of games troublesome or even impossible. The maximum tick rate of the system can be accessed with the `UserHal::TickPeriod` method, which gives the tick period in a platform-independent way. The method is introduced in e32hal.h header file. A class diagram of the Symbian OS's timer classes can be seen in Figure 3.1.

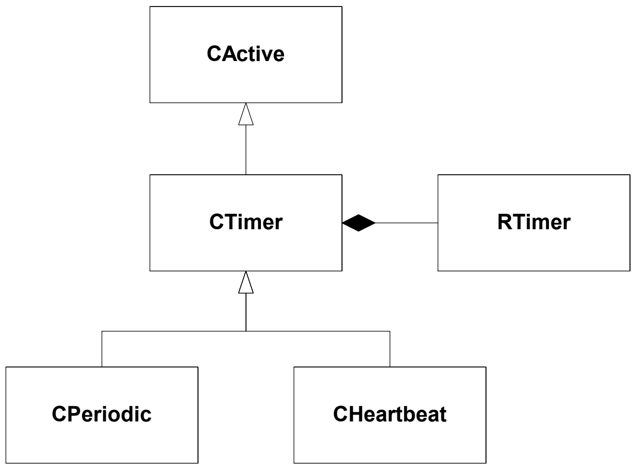


Figure 3.1 Class diagram of the Symbian OS timing services

The kernel side timer can be reached using the `RTimer` class, which is a handle to a system side server. It provides a simple API to request three different timing events: an event after a given period of time, an event at a given time, and an event which completes at a given fraction of a second. The APIs require `TRequestStatus` to be passed as a parameter which commits application developers to use active object as event handlers. To facilitate the usage of the `RTimer`, Symbian OS provides an abstract active object, `CTimer`, which encapsulates the use of the `RTimer`. This is done using a simple encapsulation where application developers need to derive from the `CTimer` and override the `RunL` method, which is called when a request is completed. However, due to the usage of active objects in timing services, the actual timer event handling may be delayed. When a timer request completes, another active object may already be running, and the active object which is handling the timer events will not be scheduled until the other active object has concluded its `RunL`. This can not be avoided but the impact on timing accuracy can be minimized by making all `RunL` methods as short-running as possible. The event handling may also be delayed if another active object with a higher priority is scheduled first. This can be avoided by making the active objects, which are handling timer events, higher priority than other active objects.

Symbian OS also provides two `CTimer` derived classes to get timer events repeatedly; `CPeriodic` and `CHeartbeat`. Both of these classes call a callback method when an event occurs. For the `CPeriodic`, the interval of the events can be given in microseconds and for the `CHeartbeat` the interval can only be given in fractions of a second, which are defined by `TTimerLockSpec` enumeration. The minimum fraction is one-twelfth. In `CPeriodic`, the given interval is rounded upwards to the closest system tick resolution. The `CHeartbeat` provides a method to synchronize the timer with the system timer. Its callback method `Synchronize` gets called if one or more timer events are missed, and this way it provides an application the possibility to perform needed recovery actions. All of the timer classes referred to above can be found from e32std.h and e32base.h header files.

3.5 Key event Handling

Symbian OS is a event driven system - all applications and servers can be seen as event handlers. The events, such as key events are handled with active objects, making the event handling non-preemptively scheduled. An example of an event flow when a user presses a key can be seen from Figure 3.2.

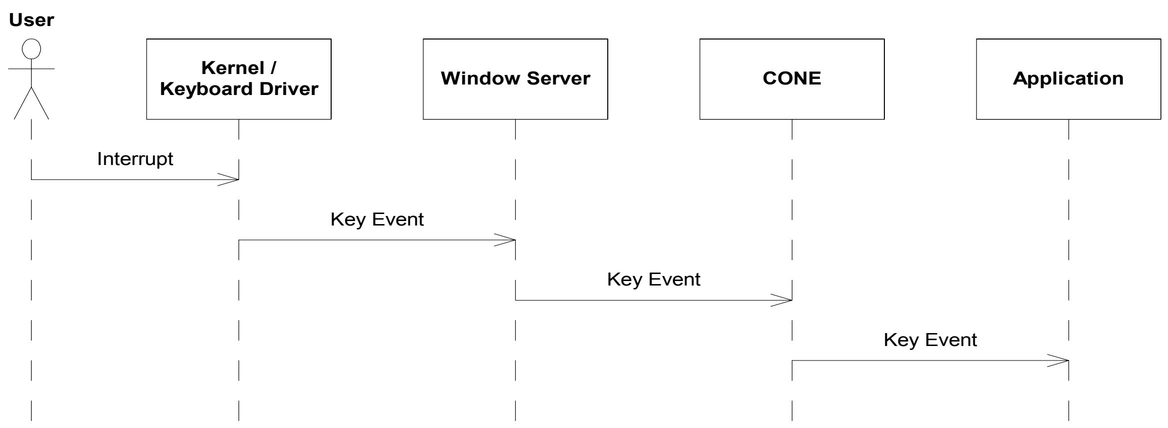


Figure 3.2 Key event flow

When a user presses a key, the keyboard hardware generates an interrupt, which is captured by the keyboard driver. After resolving the key code of the event, the driver sends it to a system side thread called window server. The window server sends the event to the application whose window group has the focus. This is done using a control environment (CONE), which is an API between the window server and a user interface library. The CONE and the window server are explained in Chapter 4.

In the application side the key events are handled in the `OfferKeyEventL` method which is called by the window server. Each key press generates three separate events. The First event is `EEventKeyDown,` which is generated when a key is pressed down. This is followed by `EEventKey`, and when the key has been released, by `EEventKeyUp`. The event types are specified by the `TEventCode` enumeration, which is passed to `OfferKeyEventL` as the second parameter. The first parameter is a struct, `TKeyEvent,` which specifies more detailed information on the event. If a key is kept down longer than 0.8 seconds, the window server sends another `EEventKey` event to the application; a long key press event. If the key is kept down longer than that, the window server sends key repeat events in every 0.25 seconds. These time frames are default values for Series 60, and they can be changed by applications.

`TKeyEvent` has a member variable, `iRepeats,` which can be used to separate a long key press from key repeat events. In case the variable differs from zero, the application needs to know what the value was when the last key event was received. If the last event's `iRepeats` equals to zero, a long key press was received and in the other case a key repeat event was received. The `iRepeats` variable is a 32-bit signed integer which defines the number of events since the last handled event. Because the most of the key events are handled somewhere, the variable does not define the actual number of repeats since the first key event. This is why applications need to count the repeats by themselves if they want to know how long the key was pressed down. The definitions of `TKeyEvent` and `TEventCode` can be found from w32std.h header file.

Games, which need key events more frequently, should set their own key repeat rates. The key repeat time frames can be changed using the window server's `SetKeyboardRepeatRate` API which takes two parameters. The first parameter specifies the time before the first key repeat event, which equals to a long key press, and the second parameter specifies the time between subsequent key repeat events. Setting the time frames equal results a linear repeat rate where time frames are equal between the first key event and subsequent ones. Because the repeat rates are system-wide settings they should be changed back to the defaults when another application is brought into the foreground.

In Series 60 most of the keys are blocked by default; only a power key and an edit key are non-blocked keys. Anyhow, key overlapping is very essential for games where a user should be able to press two keys simultaneously. This is why Series 60 provides API for disabling key blocking. The base class of application UIs, `CAknAppUi`, provides

SetKeyBlockMode method, which can be used to disable key blocking. The API takes a TAknKeyBlockMode enumeration as a parameter, which can have two possible values: EDefaultBlockMode, and ENoKeyBlock. Key overlapping is also a system wide setting which should be restored to default value when the game is not on the foreground.

3.6 Sounds

In Symbian OS, playing and manipulation of sounds is handled by the media server. The media server supports various audio file formats, such as wav, au and wve, and provides an API for applications to developed additional file format plug-in modules for the server. Media server's client API is devided into three different interfaces: audio sample editor, audio tone player and audio sample player. The audio sample editor interface provides advanced audio manipulation methods, which can be used for recording, editing and playing sounds. The audio tone player interface enables applications to create and play synthesised sounds. The audio sample player interface can be used to playback sample data files. The use of the media server interfaces requires an active scheduler to be running in the same thread.

For most of the games the audio sample player interface provides all needed features to implement desired sound effects. The interface consists of MMdaAudioPlayerCallback and CMdaAudioPlayerUtility classes. The MMdaAudioPlayerCallback is a mixin class that provides callback methods to notify a client class that an initialization or playing of a sample has been completed. This is why the class, that is using the sample player interface, needs to be inherited from the mixin class. The CMdaAudioPlayerUtility class provides methods to load and play a sample, and to set volume of the playback. The class can only be associated to a single sample data and thus an application needs to create as many instances of the CMdaAudioPlayerUtility class as it has different sample data files. Following code shows an example of the use of CMdaAudioPlayerUtility class.

```
// Create a sample player and load a sample from a file
CMdaAudioPlayerUtility* samplePlayer =
       CMdaAudioPlayerUtility::NewFilePlayerL(
                            KSampleFileName, *this );
// Play the sample
samplePlayer->Play();
```

In Series 60, every application has a default sound for each key. The sounds may also depend on whether the key press was a short, long or a repeated key press. Series 60 application UI class, CAknAppUi, provides the support for applications to specify their own key sounds in a resource file:

```
RESOURCE AVKON_SKEY_LIST r_example_skey_list
   {
   list =
          {
          AVKON_SKEY_INFO { key=EStdKeyLeftArrow;

                                 sid=EAvkonSIDNoSound;},
          AVKON_SKEY_INFO { key=EStdKeyLeftArrow;

                                 sid=EAvkonSIDNoSound;

                                 type=ESKeyTypeLong;},
          AVKON_SKEY_INFO { key=EStdKeyLeftArrow;
                                 sid=EAvkonSIDNoSound;
                                 type=ESKeyTypeRepeat;}
          };
   }
```

Available sound ids, SIDs, are specified in the avkon.hrh header file. In games, if a key is kept down for a long period of time, the repeat sound should be disabled by specifying the key event's sound ID to `EAvkonSIDNoSound`. This is because playing the repeat sound every time a key repeat event is received consumes a lot of processing time. If a continuous sound is needed by a game, the audio sample player should be used instead.

3.7 Installation

In Symbian OS, installation of applications is done using installation files, sis files. Sis files contain the files to be installed and the needed information to do the installation. The data in sis files is compressed to save memory and to minimize the time that is needed to transfer the sis files to a terminal. The installation of an application can be done directly from a PC, that has a Series 60 PC Suite installed, by running the corresponding sis file. Sis files can also be installed by first downloading the file using various communication technologies, such as WAP, Bluetooth and Infrared Data Association (IrDA), and then by opening it in a messaging application.

Sis files are constructed using package files, pkg files, which hold the required information to assemble a sis file:

```
; MyGame.pkg
;   Specifies an installation file for MyGame
;Languages
&EN
;Header
#{"MyGame"},(0x1000ABCD),1,0,0
; Required line for Series 60 devices. (Added by NOKIA)
(0x101F6F88), 0, 0, 0, {"Series60ProductID"}

"\epoc32\release\thumb\urel\MyGame.app"-
          "!:\system\apps\MyGame\MyGame.app"
"\epoc32\release\thumb\urel\MyGame.rsc"-
          "!:\system\apps\MyGame\MyGame.rsc"
```

All lines, which are preceded by a semi-colon, are comment lines. The first non-comment line specifies the supported language variants. A sis file may contain more than one language variant, although only one variant is installed at a time. The second line is reserved for a package header, that specifies the name and the ID of the application, major and minor version numbers and a build number. After that Series 60 Product Uid is stated. This indicates on which Series 60 Platform versions and devices this application can be installed. Multiple Series 60 Product Uids can be used. Here are some of the most common Series 60 Product Uids:

Nokia 7650	0x101F6F87
Nokia 3650	0x101F7962
Nokia N-Gage™ Mobile Game Deck	0x101F8A64

SX1	0x101F9071
Series 60 Platform v0.9	0x101F6F88
Series 60 Platform v1.0	0x101F795F

(previous 3 sentences and the table added by NOKIA)

The following lines define which files will be installed. Each line specifies the source path in a PC and the target path on the terminal. If the target drive letter is specified as an exlamation mark, a user may choose the drive at the intallation time. The package file format supports also some optional parameters, which can be used, for instance, to specify language dependent files. The sis files are assembled using a command line tool called makesis, which takes the corresponding pkg file as a parameter.

4. Graphics

One of the most important features from a game developer's point of view is the graphics support of an OS. This chapter describes the architecture and components of the Symbian OS's graphics support and explains how they can be used. This chapter is mainly based on references [Sy99] and [Ta00].

4.1 Graphics Architecture

The graphics support of Symbian OS is specified in the system's graphics device interface (GDI). The GDI defines drawing primitives and provides functions for drawing text, divergent shapes and bitmaps. All the system's graphic components depend ultimately on the GDI as can be seen from Figure 4.1. The components will be discussed in more detail in the following sections.

In Symbian OS drawing is performed using graphics contexts and graphics devices. The GDI provides an abstract graphics context class, `CGraphicsContext`, which is a base class for all graphics contexts. It defines drawing settings, like pen and brush style, and provides methods for applications to use GDI's graphics functionalities. The actual drawing is done in a graphics device using the settings specified in a graphics context. The base class for all device classes is `CGraphicsDevice`, which specifies the attributes of a device the drawing is assigned to. Figure 4.2 illustrates the class hierarchy of Symbian OS's graphics contexts and graphics devices.

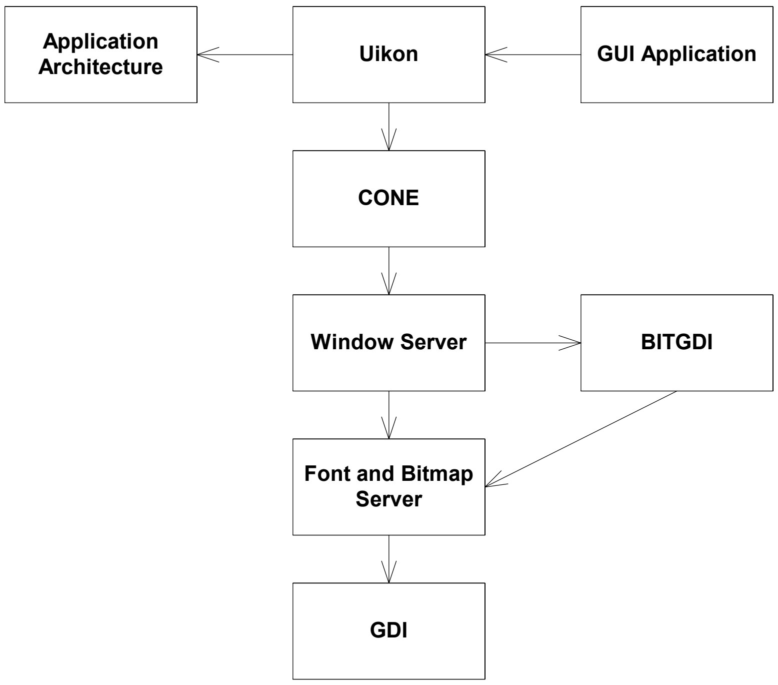


Figure 4.1 The graphics components of Symbian OS

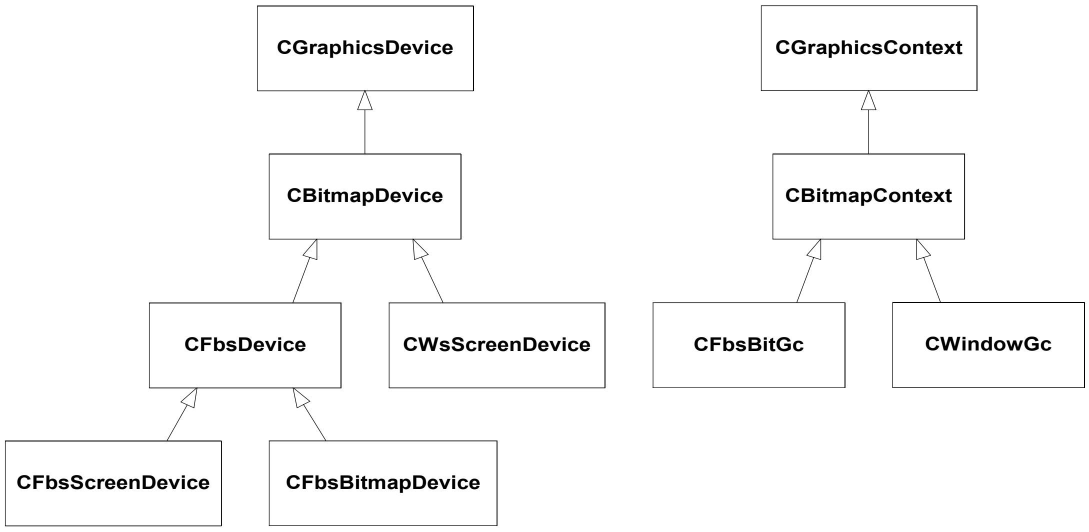


Figure 4.2 The class hierarchy of graphics contexts and graphics devices.

The concrete context and device classes are implemented in BITGDI, which is a screen and bitmap-specific graphics component. It is highly optimized with assembler code to provide fast graphics drawing. The BITGDI implements rasterising and rendering of images and it supports drawing in on- and off-screen bitmaps.

4.2 Font and Bitmap Server

The font and bitmap server's (FBS) main task is to manage fonts and bitmaps centrally, and thus allow them to be shared between all threads in the system. This allows for major memory savings as only one instance of particular data is maintained in memory.

When an application loads a font or a bitmap from the user data area, the FBS loads it to a shared heap. The server maintains a reference counter for each data item on the heap for keeping track of how many clients are using them. When the counter decreases to zero, it can be safely destroyed. The heap can be directly accessed by the window server, which eliminates time-consuming copying from the FBS's memory to the memory of the window server. All ROM-based fonts and bitmaps are used directly from the ROM.

The FBS can be accessed using the `RFbsSession` class, which is created for applications by the window server. The session class is used via the `CFbsFont` and `CFbsBitmap` classes, which provide methods for managing fonts and bitmaps. When an FBS owned data item is released by a client, the `RFbsSession` invokes a callback. This enables the window server to execute its pending redraw requests, before the data is actually deleted from the shared heap.

4.3 Window Server

The window server is used by all applications having a GUI. It provides an interface to applications allowing them to operate without direct interactions with other applications. The main task for the server is to manage system resources, like access to the screen and keyboard. This is carried out using the Symbian OS's client-server architecture which enables powerful control of shared resources. The client applications and the server run in different processes, which excludes a direct access to each other's memory address space. Thus the communication is handled using a message passing protocol. The channel between a client and the server is called a session. After the session has opened, a client may create a server request by using the session to send a message to the server. This message consists of a 32 bit request type operating code and up to four 32 bit parameters. After the request completes, the server returns a 32 bit completion code to the client. The server may also send and receive additional data using inter-process communication services. More information about the client-server architecture and the inter-process data exchange can be found from [Sy99].

Each client application communicates with the window server using a window server session class: `RWsSession`. The primary task of the class is to mediate asynchronous events to applications. Possible events are redraw events, priority key events and standard events, including user input events. The window server determines which applications and windows receive the events. For example, a keyboard event is only sent to the application whose window group has the focus, and a redraw event is only sent to the application's windows which are currently visible. Due to the higher process priority of the window server, the events are handled with higher priority than applications' other requests.

4.3.1 Client Side Buffer

Applications' requests for the window server are usually handled in the following way:

1. The window server's client side processes the request.
2. A context switch from the client process to the server process occurs.
3. The window server processes the request,
4. A context switch back to the client process occurs.

This approach ensures that the requests are handled in the right sequence, and that the requests have been processed when the control returns to the client side. However, a context switch between two processes is a quite heavy procedure, that can create big overhead for speed critical tasks. Even though, the window server has been implemented as a fixed process, which has a fixed virtual address area and hence does not need its address pointers to be updated, the context switch originates unfounded speed losses. The clients, raising a request, also has to wait for a synchronous response. For most of the requests, including drawing methods, this is unnecessary. Because of these drawbacks, the asynchronous function calls are buffered in a client side window server buffer. In the GT versions of the Symbian OS the buffer size has been fixed to 640 bytes. Series 60 has grown the buffer to 6400 bytes, and added support for applications to alter the buffer size. Larger buffer size is especially valuable in applications where drawing consists of a number of drawing functions or large amount of text. This can be seen as decreased flickering. The buffer size can be changed in the `ConstructL` of the application's UI class:

```
void CMyAppUi::ConstructL()
    {
    BaseConstructL();
    // WS buffer size can be set after the BaseContructL is called
    RWsSession &ws = iEikonEnv->WsSession();
    TInt bufsize = 10000;
    ws.SetBufferSizeL(bufsize);
    // Continue normal app UI contruction
    iMyView = new ( ELeave ) CMyMainView;
    iMyView->ConstructL( ClientRect() );
    AddToStackL( iMyView );
    }
```

When the buffer is flushed, all the methods are passed on to the window server in one message and thus only two context switches are required. The flush occurs in the following situations:

1. the buffer is full,
2. a synchronous method is called,
3. `EventReady()`, `RedrawReady()` or `PriorityKeyReady()` is called,
4. `Flush()` is called, or
5. a method which would overflow the buffer is called.

If a drawing is initiated in response to another event than a window server event, an explicit `Flush` should be called. This is the case, for example, in games, where drawing is usually initiated in response to a timer event.

4.3.2 Windows

Applications draw on the screen using windows, which are managed by the window server. Figure 4.3 illustrates the relations between different window classes.

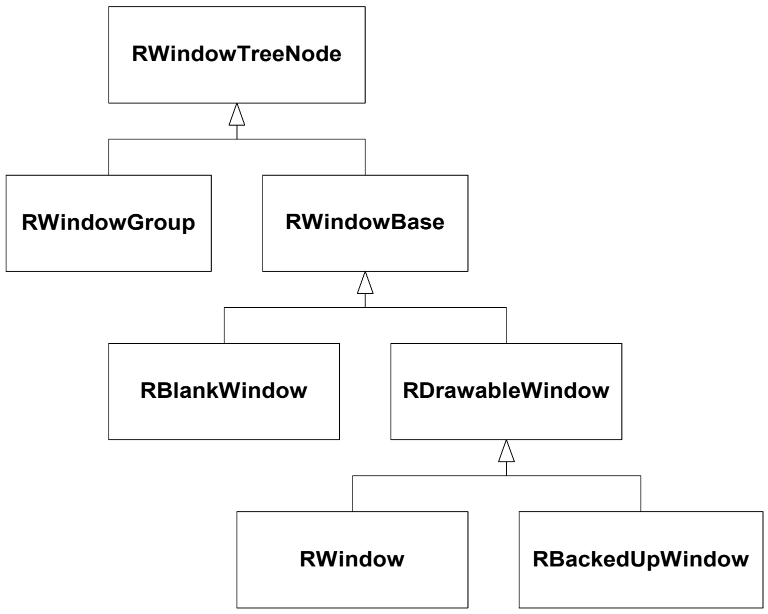


Figure 4.3 Window class hierarchy of the Symbian OS

All windows are inherited from the base class, `RWindowTreeNode`, which specifies their z-order. Displayable windows are also inherited from the abstract base class, `RWindowBase`. `RWindow` is a standard window, which can be drawn on and redraws of which are performed by the application. `RBackedUpWindow` presents a window which retains its content in a backup bitmap. When an area of the `RBackedUpWindow` gets invalid, the window server redraws it automatically from the backup bitmap, without requiring an application redraw. All classes, which are inherited from `RWindowBase`, have a display mode which specifies their colour depth. The display mode is defined by `TDisplayMode` enumeration and the mode is usually the same as the screen has. `RWindowGroup` is a non-drawable window, which can not be seen on screen. The main function of `RWindowGroup` is to handle the keyboard focus and to form a group for the applications' other windows. Typically applications have one window group which all keyboard events are associated to.

The applications' windows constitute a tree hierarchy where the upper most node is a window group. Thus every drawable window has a parent and conceivable one or more siblings and child windows. The z-order of the windows is specified by their ordinal position, which is relative to their parent window and unique among their siblings. The drawing order of windows can be changed by altering their ordinal position and an application can be brought to the foreground by altering its window groups ordinal position. The frontmost window and window group have the ordinal position of zero. The window groups ordinal position is also relative to their priority. The priority is zero by default and can be changed by applications to override the z-order specified by the ordinal position. An application whose window group has a higher priority than other applications', can not be overlapped and can not lose the keyboard focus. The APIs to access the ordinal position of a window are defined in `RWindowTreeNode`, and can be found from the header file w32std.h.

4.3.3 Control Environment

The window server provides a relatively low-level interface, which should be encapsulated in active objects due to the asynchronous services. Instead of every application implementing the communication between the window server and the application by themselves, the framework provides a control environment (CONE), which encapsulates the services provided by the window server. CONE runs in each application process and is used by all applications having a GUI. The base class of CONE, `CCoeEnv`, is an active object whose `RunL` method is invoked when an event from the window server is received. The main task of CONE is to evaluate these events, and pass them on to the correct components to be handled. Furthermore, CONE provides a cleanupstack, an active scheduler, and common utility functions for applications to use. The active scheduler is also used by CONE itself, to manage the asynchronous services from the window server. Keyboard events are defined to be handled with a higher priority than redraw events, which makes the application more responsive for user inputs.

The screen is devided into windows by the window server. To devide the screen further, the CONE provides an application side control, which is the Symbian OS's basic unit of interaction. Each drawable window consists of one or more controls, which are inherited from the abstract base class, `CCoeControl`. A control that reserves a whole window, is called a window-owning control, and a control that covers only a part of a window, is a non-window-owning control, also known as a lodger control. One reason why controls were introduced, was the advantages they provide over windows. First of all, they are more compact in memory usage than windows. They also reduce the need for the client-server communication between an application and the window server, because only one event is needed to redraw a component which has only one window - despite how many controls it has. The redraw events are generated for windows and thus a component which consists of more than one window, would need more than one redraw requests. Also the logic to detect intersections, is simpler with controls.

The controls can be nested like windows. A control which contains one or more nested controls, is called a compound control. The nested controls are called component controls, and they can be compound controls themselves. All compound controls should override two methods from the base class: `CountComponentControls` and `ComponentControl`. These methods are used for indicating the control ownership to the CONE. The compound controls also need to handle the distribution of key events to the component controls.

The window server generates key events for the window group which currently has keyboard focus. As there is normally only one window group per application, it can be thought that events are passed to the application which has the focus. To pass the events to correct controls, the CONE provides a control stack which manages the channeling for the application. The control stack contains a list of controls which enquire keyboard events. The events are passed to the controls in the stack according to their priority. If several controls have the same priority, the events are passed on according to their position in the stack.

CONE can be accessed from the `CCoeControl` and `CCoeAppUi` derived classes using the `iCoeEnv` class member. The classes also provide the `ControlEnv` method, which can be used if access to those classes is available. If neither of these means can be used, there is a static function `CCoeEnv::Static()`, which returns a pointer to `CCoeEnv`. However, the latter function should be used only if no other method is accessible, due to the use of thread-local static (TLS) which is considerably slow.

4.3.4 UI Library

The CONE itself does not provided any concrete UI components, which are provided by the UI libraries. Series 60 provides a UI library, Avkon, that has been extended and modified from the standard Symbian OS UI library, Uikon. Avkon provides various UI components which have been especially designed for the screen size of Series 60. The components have also been designed to be easily used with an ITU-T keypad.

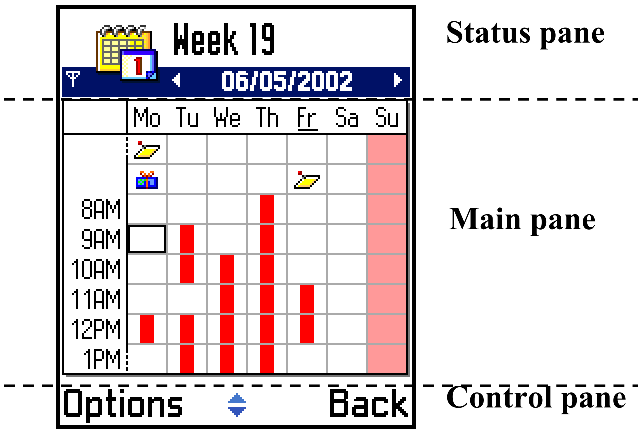


Figure 4.4 The Series 60 UI has been divided into three panes.

Avkon UI components have been divided into three categories according to their location on the screen: status pane, main pane and control pane components. Different panes are illustrated in the Figure 4.4. The status pane consists of several sub-panes: signal pane, context pane, title pane, battery pane, uni indicator pane and navi pane. Each application owns one instance of the status pane, which is automatically created on an application start up. The signal pane, battery pane, and the uni indicator pane are owned by a system side server, and they cannot be altered by applications like the rest of the sub-panes. The main pane is the area where application data is normally displayed. The Avkon UI library provides various components, and applications may implement they own concrete controls for the main pane. The components provided by the Avkon UI library include: list boxes, grids, form and different kinds of pop-up windows, such as queries and options menu. The control pane consists of softkeys and a scrolling indicator.

Even though the size of the main pane is adequate for normal applications, most of the games need as large an area as possible for their graphics. Larger game view enhances playability and clarifies game graphics. To extend a game view to cover the whole screen, games can call the `CCoeControl::SetExtentToWholeScreen` method for a control in the main pane. The status pane and the control pane can be hidden by making them invisible. This can be implemented by calling the `CCoeControl::MakeVisible` method with `EFalse` as the parameter.

4.4 Bitmaps

Symbian OS can be considered as a bitmap-oriented OS. Bitmaps are used by all applications, especially games. Even though the graphics, which are drawn using primitive drawing methods, such as `DrawLine` and `DrawEllipse`, are much more compact, bitmaps offer more efficient on-screen drawing and more detailed outcome.

Symbian OS provides its own bitmap file format, MBM, which is a multi-bitmap file. MBMs are created from Windows bitmaps using a bitmap converting tool bmconv. As a MBM file may contain more than one bitmap, the bmconv creates also a bitmap header file, MBG file, which provides an enumeration of bitmap IDs for accessing the bitmaps. When a bitmap is loaded from a MBM file, the corresponding header file should be included and the correct ID should be given as a parameter for a bitmap loading method. Bmconv can be used from a command line, or the bitmaps can be defined in a project file:

```
START BITMAP [target-file]
HEADER
TARGETPATH [targetpath]
SOURCEPATH [sourcepath]
SOURCE [colour-depth] [source-bitmap]
END
```

Bmconv can create two different types of Symbian OS bitmaps, ROM and non-ROM bitmaps. The non-ROM bitmaps, also known as file store bitmaps, are compressed using run-length encoding (RLE) and they need to be loaded in RAM before they can be used. To enhance drawing speed, the ROM bitmaps are not compressed, and thus they can be used directly from the ROM. By default bmconv creates file store bitmaps.

Symbian OS provides support for transparent bitmaps using masks. Masks are black and white bitmaps, where white colour describes the transparent area - only the pixels which are black in the mask are drawn from the original bitmap. Because masks need only two colours, they should be converted into 1 bit bitmaps to save memory. Figure 4.5 illustrates an example of the mask usage.

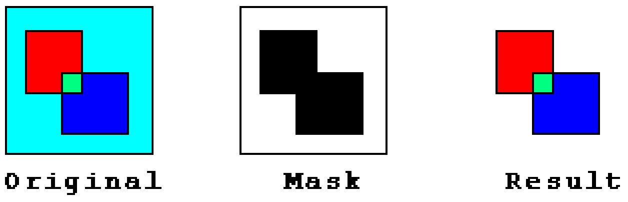


Figure 4.5 Example of using a mask to draw a transparent bitmap.

For creating masks, Series 60 provides a command line tool called makemask, which creates 1 bit masks from 8 bit bitmaps. Makemask uses the last palette index in the original bitmap as a transparent colour.

Even though Symbian OS provides some APIs to set palettes for bitmaps, they are not implemented. The APIs were supplemented before any support for colour displays was implemented. When the support was added, Symbian OS decided only to support Netscape colour cube set for its palettes. Series 60 has changed the functionality by providing its own fixed palette for 8 bit bitmaps. Bmvconv is altered to convert 8 bit bitmaps to use the Series 60 palette, which provides 216 colours and 10 grey shades. This prevents 3rd party developers to use their own palettes, which most often degrades the outcome of a bitmap on the screen. Especially the bitmaps which need numerous shades of a colour, for example to create a gradient, should be converted into at least 12 bit bitmaps. The Series 60 palette is defined in thirdpartybitmap.pal palette file.

Bitmaps are managed with a `CFbsBitmap` class, which provides methods for creating and loading bitmaps, and defines their colour depth and size. It uses the `RFbsSession` class to access the FBS and thus hides the session class from the user. The `CFbsBitmap` also provides methods to directly access the image data of the bitmap. A pointer to the data address can be obtained with the `DataAddress` method, and a specific scan line can be accessed using the `GetScanLine` method.

Bitmaps are devided into two different heaps in the FBS, according to their sizes. The bitmaps which are larger than 4 kB are stored in a different heap than the smaller bitmaps. The partition was implemented to prevent fragmentation, when large bitmaps are created and destroyed frequently - the heap for large bitmaps is automatically defragmented. Due to the defragmentation, the heap needs to be locked when a large bitmap's content is manipulated. To prevent simultaneous defragmentation and manipulation, the `TBitmapUtil` class provides operations to lock and unlock the heap. The locking of the heap needs to be done only when a bitmap's image data is edited directly – drawing and copying methods provide locking automatically. The following example locks the FBS heap if the bitmap is a large bitmap and fills in the bitmap data with colours from zero upwards. It assumes that the bitmap uses 16 bits for each pixel. This holds true for 12 and 16 bit bitmaps.

```
// Lock the heap if a large bitmap
if ( bitmap->IsLargeBitmap() )
   {
   TBitmapUtil bitmapUtil( bitmap );
   bitmapUtil.Begin( TPoint(0,0) );
   }
// Edit bitmap
TSize bitmapSize = bitmap->SizeInPixels();
TUint16* bitmapData = (TUint16*)bitmap->DataAddress();
TUint16 colour = 0;
for ( TInt y = 0; y < bitmapSize.iHeight; y++ );
   {
   for ( TInt x = 0; x < bitmapSize.iWidth; x++ )
          {
          *bitmapData++ = colour++;
          }
   }
// Release the heap
if ( bitmap->IsLargeBitmap() )
   {
   BitmapUtil.End();
   }
```

To make the drawing of bitmaps even faster than with `CFbsBitmaps`, the window server provides its own bitmap class, `CWsBitmap`. It eliminates additional context switches between the window server and the FBS by taking the ownership of a bitmap handle. The `CWsBitmap` is inherited from the `CFbsBitmap`, and it implements all the same methods. `CWsBitmap` should be used instead of its base class, when drawing speed is crucial.

4.5 Drawing

Applications draw on the screen using windows. This is done using the `CWsScreenDevice` graphics device, which is associated to the `CWindowGc` graphics context. CONE provides an instance of `CWindowGc` for applications as the standard graphics context for drawing controls. It is created by `CCoeEnv` and can be accessed using the `CCoeControls::SystemGc` method. The drawing methods of `CWindowGc` are buffered on the client side window server buffer.

Drawing can be either a system- or an application-initiated transaction. System–initiated drawing is triggered when a window is created, or when the contents of the window become invalid due to the overlapping of windows. For the latter case, the window server maintains an invalid region for every window. If a window needs to be redrawn, the window server sends a redraw event to the application which owns the invalidated window. CONE then uses the invalid region to establish the controls which need to be redrawn, and calls their Draw method. This is why every control should implement the Draw method to redraw themselves. The default implementation of the Draw in `CCoeControl` leaves the control blank. The following code shows an example of a Draw method:

```
void CExampleControl::Draw( const TRect& /*aRect*/ ) const
```

```
        {
        // Get the system graphics context
        CWindowGc& gc = SystemGc();
        // Set drawing settings
        gc.SetBrushStyle( CGraphicsContext::ESolidBrush );
        gc.SetBrushColor( KRgbRed );
        // Draw
        gc.DrawLine( TPoint(10,10), TPoint(30,10) );
        }
```

The `TRect` parameter of the `Draw` method indicates the invalid region that needs to be redrawn. Most of the controls, however, ignore the rectangle due to the fact that it is much simpler and not much slower to draw the whole control again.

Application-initiated drawing is needed when an application's data or state changes, and the screen needs to be updated. `CCoeControl` provides the non-virtual `DrawNow` method, which indicates the window server that the control is about to draw, calls control's `Draw` method, and finally indicates the window server that the control has finished drawing. `CCoeControl` also provides `DrawDeferred` method, which invalidates the window and thus originates a new redraw event from the window server. The difference between these two methods is, that `DrawNow` enforces the control to redraw itself immediately, whereas `DrawDeferred` causes a redraw event, which will be handled with a lower priority. Because the CONE handles user input events with a higher priority than redraw events, any pending user input events are handled first. These methods, however, are gratuitously heavy operations because they redraw the whole control again. Usually only the changed parts need to be redrawn, which can be done with the following code:

```
void CExampleControl::DrawBitmap( const TPoint& aPoint,
                                  const CFbsBitmap* aBitmap )
        {
        // Get the system graphics context and control rectangle
        CWindowGc& gc = SystemGc();
            // Establish drawing rectangle
        TRect rect = TRect( aPoint,
                         TSize( aBitmap.iWidth, aBitmap.iHeight ) );
        // Activate graphics context
        ActivateGc();
        // Invalidate window
        Window().Invalidate( rect );
        Window().BeginRedraw( rect );
        // Draw a bitmap
        gc.DrawBitmap( aPoint, aBitmap );
        Window().EndRedraw();
        // Deactivate graphics context
        DeactivateGc();
        }
```

The example code above draws a `CFbsBitmap` to the place defined by the `aPoint` parameter. Notable in the example is that the graphics context needs to be activated before it can be used, and deactivated after the drawing has ended. Also the window server needs to get the information that a client is about to start redrawing. This is done with the `BeginRedraw` method. The `Invalidate` method is needed, because the window server only allows an application to draw in an invalid area. In a system-initiated redraw, the CONE activates the graphics context and calls the `BeginRedraw` method for an application. The `Invalidate` method does not have to be called either because the window has already been invalidated – that is why the system initiated the redraw in the first place.

4.5.1 Sprites

A sprite is a masked bitmap, which can be moved without applications having to redraw the underlying window. Hence the games that do not need to update their background frequently can benefit from the use of sprites. This applies, for example, to PacMan like games, where some animated figures are moved on top of a non-scrollable and constant background. The redraw is performed by the window server, which makes it a high priority task which is implausibly pre-empted. This allows for smooth animation and movement of a sprite. Symbian OS provides two different types of sprites: pointers and animated bitmaps. Figure 4.6 illustrates the hierarchy of sprite classes.

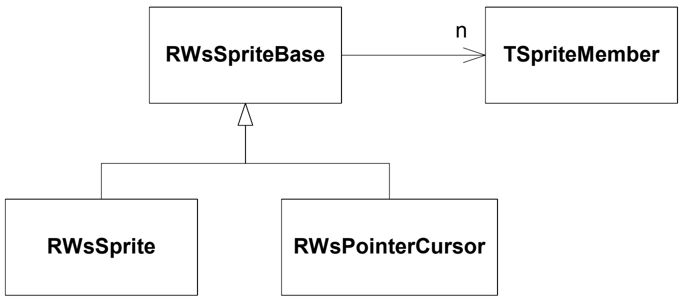


Figure 4.6 The hierarchy of sprite classes

`RWsSpriteBase` is an abstract base class for sprites. It owns one or more `TSpriteMembers`, which contain the bitmap data of the sprite. By specifying multiple members with different bitmaps, the sprite can be animated. `TSpriteMember` also defines the mask of the bitmap, the location of the bitmap within the sprite and the time interval for which the bitmap is displayed. `RWsSprite` is a concrete class for sprites. In addition to constructors, it provides only one method, `SetPosition`, which can be used to move the sprite. The following code demonstrates an example of creating a sprite using bitmaps loaded from a MBM file.

```
RWsSprite sprite = RWsSprite( iEikonEnv->WsSession() );
User::LeaveIfError( sprite.Construct( Window(), TPoint(0,0), 0 );
for ( TInt i=0; i < 8; i += 2 )
       {
       iMember[i/2].iBitmap = new ( ELeave ) CFbsBitmap();
       User::LeaveIfError( member.iBitmap->Load( KBitmapFile,
                                                i, EFalse ) );
       iMember[i/2].iBitmap = new ( ELeave ) CFbsBitmap();
       User::LeaveIfError( member.iMaskBitmap->Load( KBitmapFile,
                                                i+1,  EFalse  )
                                     );
       iMember[i/2].iInvertMask = EFalse;
       iMember[i/2].iOffset = TPoint(0,0);
       iMember[i/2].iInterval = TTimeIntervalMicrosecond32(100000);
       User::LeaveIfError( sprite.AppendMember( iMember[i/2] ) );
       }
```

After the sprite members have been updated and appended to `RWsSprite` class, the sprite can be activated by calling `RWsSpriteBase::Activate`. After this the sprite is displayed on the screen and it is ready to be moved. The content of the sprite can be changed using the `RWsSpriteBase:UpdateMember` method. Because `CFbsBitmaps` are also accessible to the window server, only the bitmap handles of a sprite are sent to the window server. This makes updating of a sprite's bitmaps considerably fast. When a sprite is no longer needed, the window server resources need to be released by calling `RWsSpriteBase::Close`. This does not free the client side member data, which needs to be deleted. `RWsPointerCursor` is a class for applications to create cursors. Its use is similar to the `RWsSprite` class, with the exception that after a pointer has been activated, it will not be shown on the screen until `RWindowTreeNote::SetPointerCursor` is called.

4.5.2 Double buffering

If a game's graphics consists of multiple moving objects which need to be updated frequently, the window server's client side buffer may be filled up and thus be flushed before all objects have been updated. This may appear as flickering for the user. Flickering or other undesirable effects may also occur if a view, that is built up over a long period of time, is drawn while it is still been updated. A solution for these problems is to use double buffering, where graphics are first drawn in an off-screen bitmap, which is then drawn on the screen as one single window server operation. Especially games which redraw their screens several times in a second, can benefit prominently from the use of an off-screen bitmap.

An off-screen bitmap can be created using bitmapped graphics context and graphics device classes; `CFbsBitGc` and `CFbsBitmapDevice`. They are created and used similarly with other context and device classes. To achieve additional performance, the bitmap itself should be a `CWsBitmap` bitmap. After the off-screen bitmap has been updated, it can be drawn in the window using normal window server's drawing methods.

When an application draws a bitmap in a window, it gets converted into the same display mode than the window. This is a time consuming operation that can slow down the drawing substantially. Hence the games which, for example, use bitmaps for their animations, should do the conversion before the animation is started. The conversion can be carried out by using an off-screen bitmap as the following example method demonstrates:

```
CFbsBitmap* CExampleControl::LoadAndConvertBitmapL(
                                Const    TDesC&    aFileName,    TInt
                                aBitmapId )
       {
       // Load the bitmap
       CFbsBitmap* originalBitmap = new ( ELeave ) CFbsBitmap();
       CleanupStack::PushL( originalBitmap );
       User::LeaveIfError( originalBitmap->Load( aFileName,
                                               aBitmapId,  EFalse  )
                                          );
       // Create a new bitmap, graphics device and context
       CFbsBitmap* newBitmap = new ( ELeave ) CFbsBitmap();
       CleanupStack::PushL( newBitmap );
       newBitmap->Create( originalBitmap->SizeInPixels(),
                                 Window()->DisplayMode() );
       CFbsBitmapDevice* graphicsDevice = CFbsBitmapDevice::NewL(
                                          bitmapConverted );
       CleanupStack::PushL( graphicsDevice );
       CFbsBitGc* graphicsContext;
       User::LeaveIfError( graphicsDevice->CreateContext(
                                               graphicsContext ) );
       TPoint zero(0,0);
       // Blit the loaded bitmap to the new bitmap
       bitmapContext->BitBlt( zero, originalBitmap );
       CleanupStack::Pop(3);
       delete bitmapContext;
       delete bitmapDevice;
       delete originalBitmap;
       return newBitmap;
       }
```

The example method gets a filename and a bitmap ID as parameters, and loads the corresponding bitmap from a MBM file. To convert the bitmap to the window's display mode, a new bitmap is created and the loaded bitmap is blitted into it. If a game has multiple bitmaps which should be converted, the conversion should be done at the initialisation stage of a game or a level. Hence the operation can be hidden from a user.

4.6 Direct draw

Drawing on the screen, using the window server, requires a context switch, which decelerates drawing speed. To bypass the window server, and thus to get rid of the context switch, a game can access the screen directly. This is called direct drawing. In Symbian OS there are two ways to draw directly on the screen.

`CFbsScreenDevice` is a graphics device that can be addressed to a screen driver, SCDV.DLL. After creating a `CFbsBitGc` graphics context for it, it can be used like any other graphics device. However, the drawing is done directly on the screen without using the window server. The other way to draw directly on the screen, is to enquire screen memory address from the system. This can be done using the `UserSrv` class:

```
TPckgBuf<TScreenInfoV01> infoPckg;

TScreenInfoV01& screenInfo = infoPckg();

UserSvr::ScreenInfo(infoPckg);

TUint16* screenMemory = screenInfo.iScreenAddress + 16;
```

The screen memory has a 32 byte header, which needs to be taken into account when writing to the memory.

Even though writing directly on the screen memory is slightly faster than the `CFbsScreenDevice`, the functionality may differ depending on the hardware and screen's device drivers. In some Symbian OS based terminals the screen is automatically updated from the screen memory when the memory is changed, whereas in other terminals the drawing needs to be explicitly activated. The screen memory address is also valid only on target hardware, and thus the drawing code needs to be devided into hardware and emulator dependent parts. In the emulator environment the drawing can be done to an off-screen bitmap, instead of the screen memory, which is then blitted to the screen using normal window server drawing methods. The environment can be detected by using `__WINS__` definition.

```
#ifdef __WINS__ // Emulator environment

// Draw to an off-screen bitmap

#else // Hardware environment

// Draw directly to the screen memory

#endif
```

A common problem for both of the direct draw methods, is that the window server is not aware of the drawing, and thus it can not notify an application, if another window or window group is brought forward. Even though applications get an event when they loose focus, they can not stop the direct drawing fast enough, and the screen will most probably get fouled. This may happen, for instance, when an incoming call arrives, and the telephony application is brought forward.

The late versions of the GT version 6.1, which Series 60 is based on provide an API for direct draw that will solve the problems referred to above. The API consists of two classes: a mixin class `MDirectScreenAccess`, which provides call back methods for an application, and a concrete class `CDirectScreenAccess`, which handles communication to the window server. The following code illustrates how an instance of `CDirectScreenAccess` is constructed, and how the direct draw support is activated.

```
iDrawer = CDirectScreenAccess::NewL(
                                iEikonEnv->WsSession(),
                                *iEikonEnv->ScreenDevice(),
                                Window(),
                                *this);
iEikonEnv->WsSession().Flush();
iDrawer->StartL();
iDrawer->ScreenDevice()->SetAutoUpdate(ETrue);
```

The `CDirectScreenAccess`’s `NewL` method takes a window server session, CONE’s graphics device, application’s window, and a pointer to a `MDirectedScreenAccess` derived class as parameters. Before the `CDirectScreenAccess::StartL` is called to activate the direct draw support, the client side window server buffer should be flushed. To enable automatic updating of the screen, screen device’s `SetAutoUpdate` method needs be called with `ETrue` as a parameter. When the direct draw support is activated, the `CDirectScreenAccess` creates a `CFbsBitGc` graphics context, which can be used by applications to draw on the screen:

```
iDrawer->Gc()->BitBlt( TPoint(0,0), iBitmap );
```

When another window is brought on top of the application’s window, the `CDirectScreenAccess` gets an event from the window server to abort drawing. The `CDirectScreenAccess` then calls `MDirectScreenAccess` derived class’ `AbortNow` method, which has to be overridden by an application to abort the drawing. To prevent the screen to become fouled, the window server does not draw the overlapping window until the abort drawing event is handled.

5. Communications

In this chapter the communications components of the Symbian OS are discussed. The communication technologies which are more important in a game developer’s perspective have been emphasized. The last section introduces a support for games to receive additional game data. [Ta00], [Di02], [No02a] and [Sy99] have been used as references.

5.1 Communications architecture

The mobile nature of smart phones and the development speed of communication technologies set demands for their communications modules. Smart phones are used in various places where available communication services may vary significantly. The existing services and technologies are continuously evolving and new technologies are introduced expeditiously. These facts create demands for flexibility and expandability for smart phones' communication modules, and hence the Symbian OS's communications architecture was designed with these values in mind. It consists of several smaller modules and supports plug-in modules which can be loaded at run-time. Also the communication settings can be changed without a reboot.

The version 6.1 of the Symbian OS supports multiple communication technologies, that are introduced in Figure 5.1. The available technologies of a particular smart phone depend on its hardware solutions. Nokia 7650 or Nokia 3650 (added by Nokia), for instance, does not have a serial cable, but supports serial communications via Bluetooth technologies.

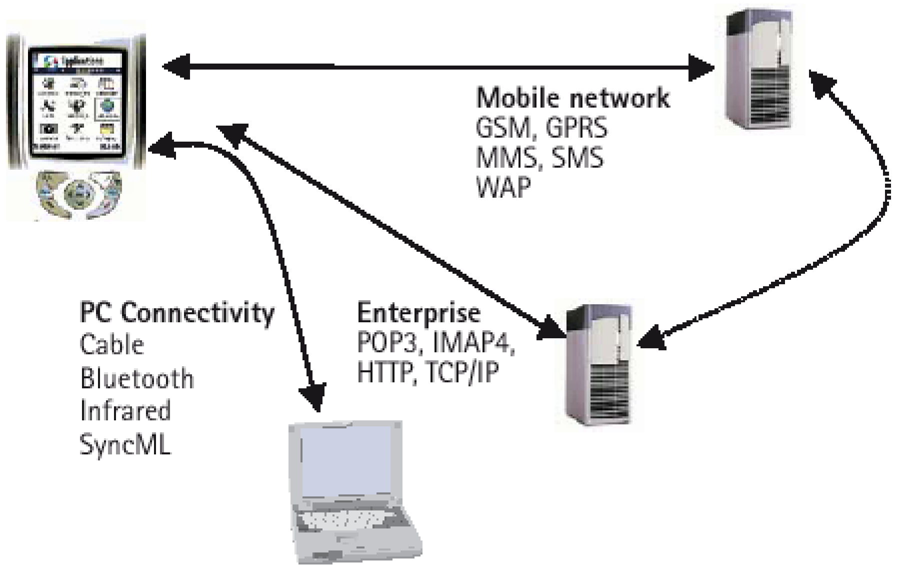


Figure 5.1 Series 60 communication technologies [No01b]

The communications architecture of the Symbian OS is based on top of three communication servers: ETEL, C32 and ESOCK. The communication services, provided by the servers, are asynchronous operations and thus they need to be encapsulated into active objects. Typically a client application creates three different active objects: one for sending data, one for receiving data and one for application side communications engine. The dependencies of communication modules are represented in Figure 5.2.

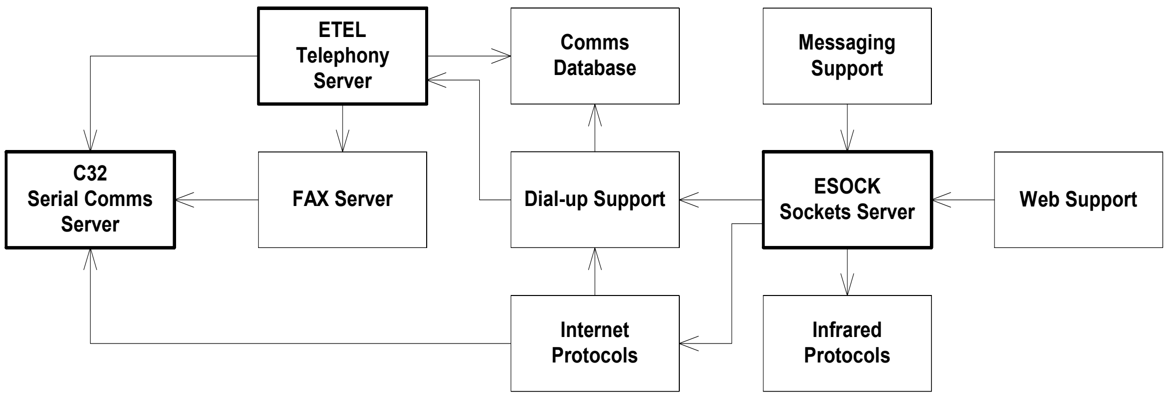


Figure 5.2 Symbian OS's communications components [Ta00].

ETEL is a telephony server which provides an interface for applications to access various telephony hardware and services, such as GSM handsets, analog modems and fax services. The server uses dynamically loadable plug-in modules, telephony server modules (TSYs), which convert the hardware specific information to application intelligible format. The client side API of the server is defined in etel.h header file, and it mainly consists of `RTelServer`, `RPhone`, `RLine` and `RCall` classes.

5.2 Serial Communications Server

Serial communications server (C32) provides a serial port API for its clients. The server uses communication server plug-in modules (CSYs), which handle the actual communication protocols. Symbian OS provides multiple CSY modules, such as the ones for handling RS232 and infrared serial communications. Custom CSY modules can also be developed by application developers using serial protocol module API, which is defined in cs_port.h header file.

The serial communications server is used similarly regardless of the used CSY module. First at an initialisation stage a client loads the needed drivers, opens the server and loads the CSY module. After the actual device has been opened and configured, the server is ready to send and receive data. Finally, all resources need to be released. The client API of serial communications server is defined in the c32comm.h header file, and it mainly consist of `RComm` and `RCommServ` classes. The following code shows an example of an initialisation stage for infrared serial communication.

```
// Load device drivers
TInt err = User::LoadPhysicalDevice( _L("EUART1") );
if ( err != KErrNone && err != KErrAlreadyExists )
       User::Leave( err );
err = User::LoadLogicalDevice( _L("ECOMM") );
if ( err != KErrNone && err != KErrAlreadyExists )
       User::Leave(err);
// Start serial communications server of type RCommServ
User::LeaveIfError( iServer.Connect() );
// Load CSY module for IrComm
User::LeaveIfError( iServer.LoadCommModule( _L("IRCOMM") ) );
// Open port of type RComm
User::LeaveIfError( iPort.Open( iServer, _L(""IRCOMM::0""),
                                  ECommExclusive ) );
```

The example code above can be changed to use RS232 by loading the ECUART CSY module, instead of the IRCOMM, and by opening `COMM:0` port instead of the `IRCOMM:0`. The `ECommExclusive` enumeration in `RComm::Open` prevents other `RComm` clients from using the port.

Writing to and reading from a port can be done with divergent `RComm::Read` and `RComm:Write` methods, which all take a `TRequestStatus` and a descriptor as parameters. When a transfer has been completed, an event is generated for the active object, whose `TRequestStatus` was passed to the transfer method. This causes the active object's `RunL` to be called. The descriptor parameter is always an 8-bit descriptor, which in consequence needs to be converted into Unicode, if the transferred data were text. The data transfer methods may also take additional parameters, which specify maximum length or time characteristics for the data transfer.

The serial communications server provides an easy way for applications to use infrared for their communications. However, the main reason for the IrComm support is to make conversion of legacy applications, that use a serial port for their communications, easier. The downside of the IrComm, compared to other IrDA protocols, is that it does not provide all IrDA services. For instance, the maximum data transfer speed with the IrComm is only 9600 bits per second, which is not fast enough for example for multi terminal games, which transfer large amount of data.

5.3 Sockets Server

The sockets server (ESOCK) provides an interface to communications protocols using sockets. The client API is generic for all protocols and the protocol-specific behaviour is specified through utility types and constants. The sockets server uses protocol modules, such as TCP/IP, IrDA and Bluetooth, that are loaded dynamically at run-time. One protocol module may contain more than one protocol. For example the IrDA module contains raw IrMUX, IrTinyTP, IrLAP, IrLMP and IrObex protocols. Common protocol modules can be develop by application developers, correspondingly to TSY and CSY modules.

The main classes in the client API of the sockets server are `RSocketServ`, and `RSocket`. Similarities to `RCommServ` and `RComm` can be seen. The `RSocketServ` handles a session for the server and provides information about available protocols, but doesn't provide any data transfer services - they are provided by the socket class `RSocket`. As the client API is same for all protocols, the properties and semantics of an individual protocol are differentiated with the `TProtocolDesc` structure. The `RSocketServer::GetProtocolInfo` method can be used to access these characteristics of the currently loaded protocol. The sockets server needs at least two different sockets to function comprehensively. One is used for listening incoming connection requests, and the other is needed for establishing a connection and transferring data. The client API of the sockets server is defined in es_sock.h header file.

One of the sockets server's most engrossing protocols from the game developers' point of view is Bluetooth. It provides a relatively fast, short range solution, that is free to use, and thus very applicable for games. Bluetooth's advantage over infrared is its longer operating radius and its ability to function without any visual contact between terminals.

Bluetooth consists of a protocol stack, which is illustrated in Figure 5.3. Symbian OS version 6.1 supports Bluetooth version 1.0, and provides applications with full access to RFCOMM, L2CAP and SDP protocols. The RFCOMM protocol simulates serial communication, and thus facilitates conversion of legacy applications to use Bluetooth. Common choice for applications is to use the logical link control and adaption protocol (L2CAP), which provides more powerful functions for applications to control the Bluetooth link. The service discovery protocol (SDP) allows applications to make enquiries for services and service providers. Usually when a new connections is created, the SDP is used to search the desired terminal, and to set up the connection settings. Symbian OS also provides a complete UI component to search the Bluetooth terminals available. It uses Symbian OS's notifier framework, which creates a dialog type component on top of applications' windows and shows available terminals as a list. The Bluetooh APIs, like other Symbian OS's communication APIs, are documented with example codes in [No02a].

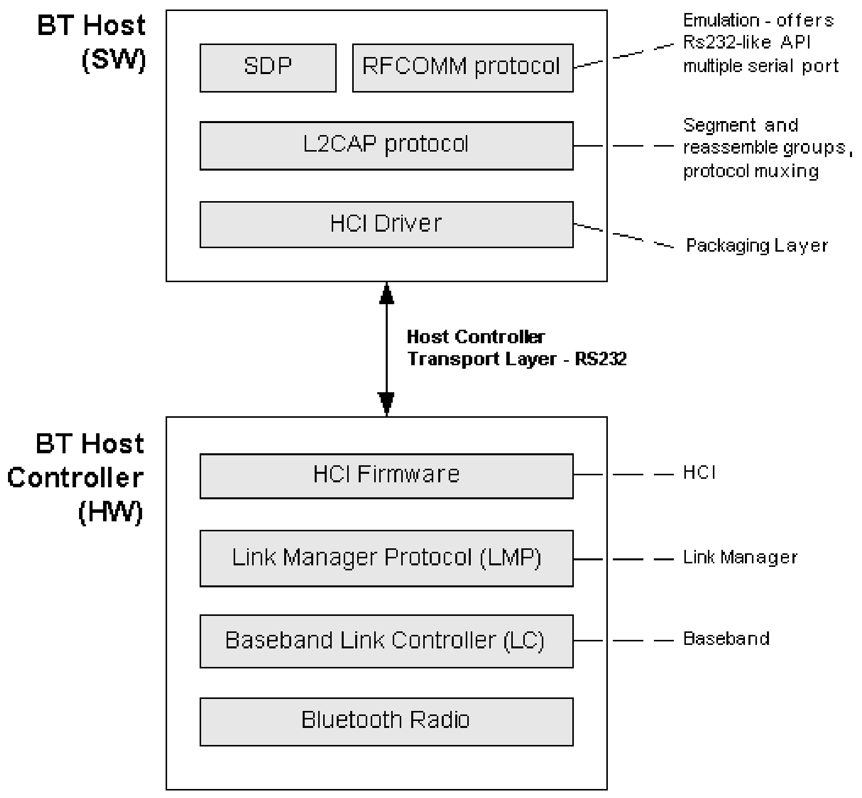

BT Host (SW)
SDP
RFCOMM protocol
Emulation - offers Rs232-like API multiple serial port
L2CAP protocol
Segment and reassemble groups, protocol muxing
HCI Driver
Packaging Layer
Host Controller Transport Layer - RS232
BT Host Controller (HW)
HCI Firmware
HCI
Link Manager Protocol (LMP)
Link Manager
Baseband Link Controller (LC)
Baseband
Bluetooth Radio

Figure 5.3 Bluetooth stack. [No02a]

5.4 Game Data Receiving

To support games to receive additional game data, Series 60 enables 3rd party games to register their data file formats in the OS, by using Multipurpose Internet Mail Extensions, MIME types. The MIME types are used by communication applications, like the WML browser and the messaging application, to find out the path where a file, having a specific type, should be saved. The files can contain, for example, new levels, weapons or graphics for a game.

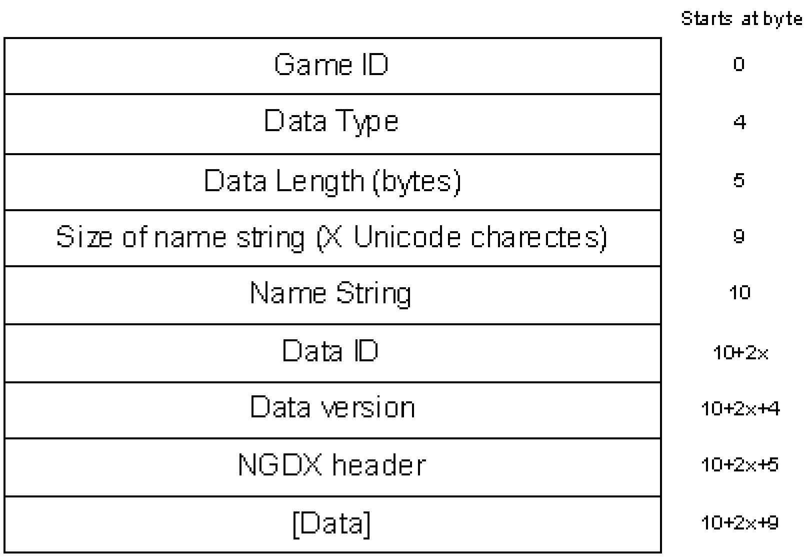
In Series 60 the MIME types for games are of the format: `application/x-NokiaGameData-<APPLICATION-ID>`, where `<APPLICATION-ID>` is the game's Symbian OS application UID's last eight digits. The MIME types are declared in Symbian OS application information files, aif files:

```
RESOURCE AIF_DATA
        {
        app_uid=0x12345678; // Application UID
        datatype_list =
                {
                DATATYPE
                        {
                        priority = EDataTypePriorityHigh;
                        type = “application/x-NokiaGame-Data-12345678”;
                        }
                };
        }
```

The priority in the `DATATYPE` structure specifies how well the current application handles the data format. `EDataTypePriorityHigh` should be used for data formats which can not be handled by other applications.

The target path for the received data file is specified in an ini file, named as `<APPLICATION-ID>`.ini. The file needs to be in Unicode format, and it contains `SDDataDir=<GAME-PATH>`, where `<GAME-PATH>` is the path for the received file. The path is relative to the default game data directory, c:\nokia\games. When a game is installed, the ini file needs to be copied to \System\SharedData directory, which can be easily done by specifying the paths in a game’s pkg file.

Series 60 specifies a standard header structure for game data files, that needs to be adhered to MIME types to work. The structure of the header format is illustrated in Figure 5.4. The Data type field can be used to specify the type of a file that is internal for the game. The Name string is an Unicode string that can be used to specify a user visible text for selecting the data item in a menu. The Data ID and the Data version are unique numbers which specify the type and the version of the file’s data. The NGDX field must contain the ASCII string “NGDX”.



	Starts at byte
Game ID	0
Data Type	4
Data Length (bytes)	5
Size of name string (X Unicode charectes)	9
Name String	10
Data ID	10+2x
Data version	10+2x+4
NGDX header	10+2x+5
[Data]	10+2x+9

Figure 5.4 Standard header for game data files [No02b]

References

[Di02] Symbian OS Communication Programming, Course material of Digia, Digia Ltd., 2002. 268 pp.

[No01a] Coding idioms for Symbian OS, ver.1.0, Nokia, 2001. 26 pp.

[No01b] Designing Applications for Smartphones - Series 60 Platform Overview, Nokia, 2001. 14 pp.

[No02a] Nokia Series 60 Software Development Kit for Symbian OS, Version 0.9, CD-ROM, Nokia Ltd., 2002 (part of this Development Kit)

[No02b] Support Guide for Developers on Receiving Game Data, ver.1.0, Nokia Ltd., 2002. 4 pp.

[Sy02] Symbian OS Version 6.x Detailed Operating System Overview, http://www.symbian.com/technology/symbos-v6x-det.html, Symbian Ltd, (6.3.2002)

[Sy99] EPOC Release 5 C++ Software Development Kit, CD-ROM, Symbian Ltd.,1999.

[Ta00] Tasker, M. et al, Professional Symbian Programming. Mobile Solutions on the EPOC Platform. Birmingham, UK: Wrox Press Ltd., 2000. 1031 pp.