Developer Platform for Series 60: Using the Core ETel APIs

Version 1.0

December 15, 2003

SERIES 60 PLATFORM

Legal Notice

Copyright © 2003 Nokia Corporation. All rights reserved.

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.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1. Introduction 5
   1.1 Purpose and Scope 5
2. ETel Overview 6
   2.1 Data, Fax, and Voice Support 6
   2.2 Standard Phone Features 6
   2.3 ETel Architecture 6
3. ETel Core API 9
   3.1 Root Server Session — RTelServer 9
   3.2 Phone Subsession — RPhone 9
      3.2.1 Finding information on phones 10
      3.2.2 Finding information on lines 10
      3.2.3 Notification of changes 10
   3.3 Line Subsession — RLine 10
      3.3.1 Opening a line subsession 10
      3.3.2 Querying the line state 11
      3.3.3 Receiving notification of changes 11
   3.4 Call Subsession — RCall 12
      3.4.1 Overloaded functions 12
      3.4.2 How to open a call 13
      3.4.3 Call ownership 13
      3.4.4 How to make a call 13
      3.4.5 How to hang up a call 14
      3.4.6 Querying a call properties 14
      3.4.7 Receiving notification of changes in the call 15
      3.4.8 How to answer a call 15
      3.4.9 Data calls 15
4. AT Commands 17
   4.1 Basic AT Command Syntax 17
   4.2 AT Commands on the Series 60 Platform 17
   4.3 Summary of AT Commands 19
5. Summary 20
6. Terms and Abbreviations 21

Change History

December 15, 2003	1.0	Initial document release

1. Introduction

1.1 Purpose and Scope

This document describes the architecture and features of the ETel Telephony implementation on the Developer Platform for Series 60, its APIs, and how to use them. It also describes the limitations of the system.

Code samples are provided throughout the document to demonstrate the key features.

2. ETel Overview

The ETel API is the interface to the telephony functionality in the Developer Platform for Series 60. It is designed to be hardware and network independent, by handling what is considered the generic telephony functionality of a telephone. ETel is built around a core set of functions that are supported by almost all telephony devices and services. Developers can use this API to ensure that their applications will work with most telephony equipment. Newer technologies and device-specific advanced functionality can be accessed via extensions to the core API.

ETel uses plug-ins called Telephony Extension (TSY) modules, which convert information between the format expected by a device (hardware-specific) and the format that applications can access vie the core ETel API.

Where noncore functionality is provided by telephony equipment, e.g., GSM, this is passed to extensions of the core API.

2.1 Data, Fax, and Voice Support

ETel supports three forms of communication: *voice*, *data,* and *fax*. While the concept of a voice call is commonly understood, a data call allows the transfer of data over the telephone system. And while ETel does support fax calls, they are not directly available to applications. The public API for fax transmissions is contained within **CFaxTransfer** and will not be considered here.

2.2 Standard Phone Features

ETel provides the ability to dial numbers, hang up calls, give alerts to the presence of incoming calls, and answer incoming calls. It further provides the mechanisms to both determine and give notification of changes in the state of the phone, line, and calls. It is possible, prior to initiating a specific telephony behavior, to query the capabilities and state of a particular phone, line, or call, to see if the functionality is supported.

Some functionality commonly considered part of telephony is not supported, either because it is more suitably implemented elsewhere, or because it has not been implemented but may be included in a later ETel version. For example, ETel does not support calling cards, telephone number localization, or message distribution.

2.3 ETel Architecture

ETel is implemented as a server, and as such its services can be accessed simultaneously by a number of applications. The ETel architecture is based on a four-tier, hierarchical system of *root session*, *phone*, *line*, and *call*.

The root session, implemented by `RTelServer`, provides an interface to the devices' telephony system and information on the system telephony and available phones.

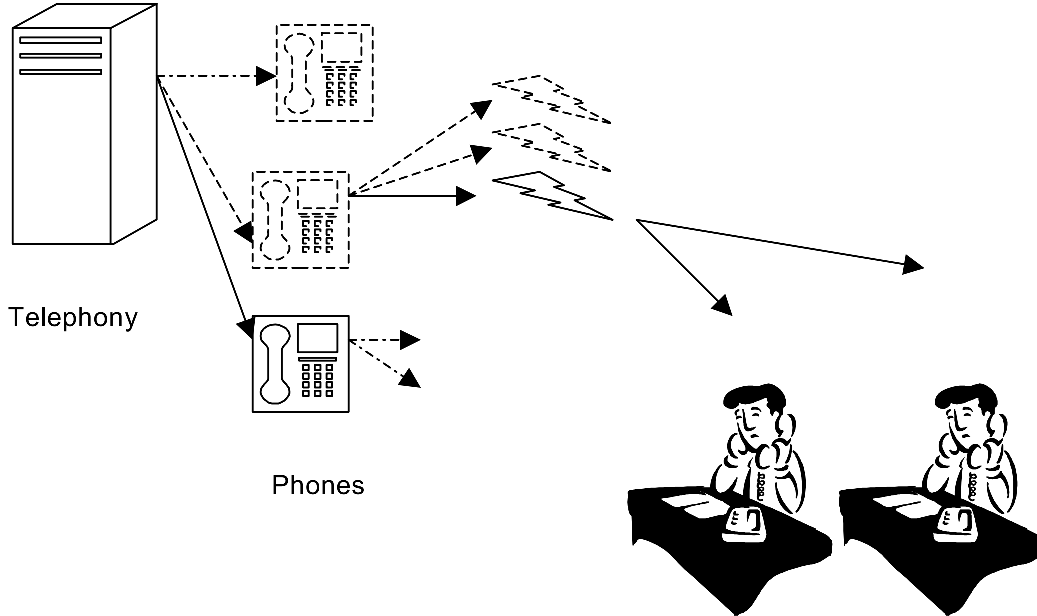

Figure 1: Server, phone, line, and call hierarchy

Once a root server session is established, the `RPhone`, `RLine`, and `RCall` classes provide a means to retrieve information concerning the state of phones, lines, and calls. Further, notification of changes of state in these objects can also be requested.

`RPhone` provides the interface to the phone subsession, abstracting a particular telephony device. A phone can have one or more lines. Generally there will be only one phone per device, although this is not a requirement.

`RLine` provides the line interface, and each phone may support multiple lines, i.e., voice, data, or fax lines. Lines can have zero or more active calls.

`RCall` implements the call interface. A call provides basic telephony functionality, e.g., call initiation, answer, and hang up.

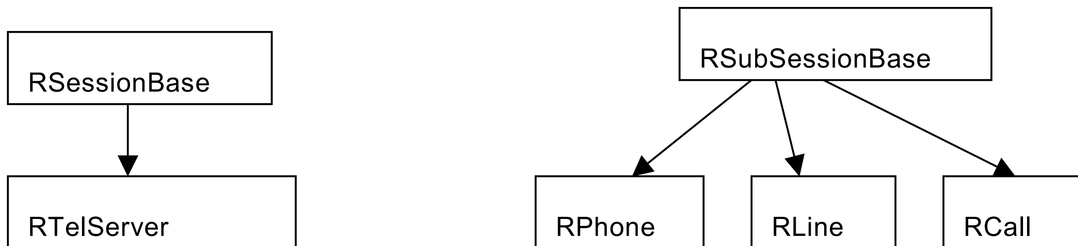

Figure 2: ETel class relationships

The translation of client-side API calls to instructions understood by a particular device is done by an ETel telephony extension module (TSY) device driver. TSYs are loaded dynamically and will normally support all of the functions in the core client-side API. An individual TSY may possibly support other functions in the client API, but there is no requirement for it to do so; TSYs may support as much or as little functionality as the manufacturer desires, allowing great control of the implementation of the device. Developers should always remember that functionality supported by a specific phone might not be available on another phone that uses a different TSY.

If a TSY does not support a called function, ETel returns `KErrNotSupported`.

	Note: The TSY used on a device is different from that used on the emulator during development. Developers should make sure they always use the correct TSY.

3. ETel Core API

3.1 Root Server Session — RTelServer

The root server session provides access to general telephony information and functionality. Clients wishing to use the ETel server must open a connection with the root server using functions defined in `RTelServer`. The interface provides functions that allow clients to load and unload TSY modules, list the number of phones that are supported by the loaded TSY modules, obtain information about the phones in the loaded TSY modules, and close an open root session.

To access functionality associated with a specific phone, the client must open a subsession for that phone.

Connection to the telephony server is achieved via `Connect()` and must be called before any other functions during a telephony session. After connecting to the telephony server, a TSY module needs to be loaded. A TSY module is a plug-in for the telephony server and is loaded using `LoadPhoneModule()`, taking a `TDesC,` which is a descriptor containing the name of the TSY to be loaded. The name of the TSY can be found either from the commsdb or by searching `\System\Libs\` for files with a `.tsy` extension. A TSY module called "PHONETSY" is provided with the Series 60 platform.

Hardware vendors may have/require device-specific TSYs tailored for their particular needs.

Once the TSY module has been loaded, information about the phones added by the TSY can be found using **EnumeratePhones()** and **GetPhoneInfo()**.

- **EnumeratePhones()** returns the total number of phones supported by all the loaded TSYs.
- `GetPhoneInfo()` gets information about a particular phone, filling in a `TPhoneInfo,` which includes the phone name and the number of lines the phone supports.

A TSY module can be unloaded explicitly using `UnloadPhoneModule()` and is unloaded automatically when the `RTelServer` session is closed. Information about the currently loaded TSY module can be found using `GetTsyName()` and `GetTsyVersionNumber()`. The extended funtionality, if any, that the loaded TSY supports can be found using `IsSupportedByModule()`. It is assumed that the TSY module supports all functionality defined in the core API.

3.2 Phone Subsession — RPhone

The phone subsession, `RPhone`, provides access to the functionality associated with a specific phone. To open an `RPhone` session, `Open()` takes two parameters, the first being a reference to an already opened `RTelServer` session, the second being the name of the phone subsession to be opened. The name of the phone subsession can be retrieved by calling `RTelServer::GetPhoneInfo()` and using the `iName` member variable of the returned object. The modem is automatically initialized when it is opened. The subsession opened must be explicitly closed with `Close()`, otherwise it will result in a memory leak.

3.2.1 Finding information on phones

Three functions are provided to query the state of the phone:

- `GetCaps()` assigns a value to a `TCaps` parameter. It contains a flag of the phone's capabilities.
- `GetStatus()` assigns a value to an `RPhone::TStatus` parameter. This parameter contains `TMode` and `TModemDetected`.
- `GetInfo()` assigns a value to a `TPhoneInfo` parameter, which contains the current modem detection state.

3.2.2 Finding information on lines

Two functions are provided allowing the client to query information about the current state of lines:

- `EnumerateLines()` returns the number of lines the phone supports.
- `GetLineInfo()` takes a `TLineInfo` parameter that contains the line's name, current status, and line capabilities flag. For example:

```
TInt count;
iPhone.EnumerateLines(count); // find number of lines
RPhone::TLineInfo lineInfo;
for(TInt i=0; i<count; i++)
   {
   User::LeaveIfError(iPhone.GetLineInfo(i, lineInfo));
         // More Code
   }
```

3.2.3 Notification of changes

`RPhone` implements two notification functions that, along with their associated cancellation functions, allow changes to be monitored:

- `NotifyModemDetected()` takes a `TModemDetection` parameter and is of little use where the modem is permanently connected to the computing device.
- `NotifyCapsChange()` takes a `TCaps` parameter that, on request completion, contains the phone's capabilities.

3.3 Line Subsession — RLine

The line subsession, `RLine`, provides access to the functionality associated with a specific line. Each phone can have one or more lines, i.e., voice, data, and fax. Information about the capabilities and status of a line can be queried, and notification of changes in these can be provided. Each line can have zero or more active calls.

3.3.1 Opening a line subsession

A line subsession can be opened from a phone subsession or from the root server session if access to phone functionality is not required:

```
TInt Open(RPhone& aPhone, const TDesC& aName);
```

```
TInt Open(RTelServer& aServer, const TDesC& aName);
```

In the second version, the name of the phone must be included in the aName parameter (PhoneName::LineName), which is why it is not recommended to use this version when doing cross-platform development. The name of the phone is usually retrieved from RPhone at runtime and should not be hard coded.

Both methods take as a parameter the name of the line to be opened, which can be found using `RPhone::GetLineInfo()`.

	Note: An open-line subsession must be explicitly closed with `RLine::Close()` before the `RLine` object is destroyed, otherwise a memory leak will occur.

3.3.2 Querying the line state

Information about the calls opened on a line can be queried:

- `EnumerateCall()` returns the number of calls opened on a line.
- `GetCallInfo()` takes an `RLine::TCallInfo` parameter that returns the name of the call, call status, and capabilities of the call. The name is needed to differentiate the call from others on the line.

The capabilities of the line, status of the line, and hook status can be found using `GetCaps()`, `GetStatus()`, and `GetHookStatus()`, respectively.

`GetInfo()` is also available; it takes a `TLineInfo` parameter, which returns with the current hook status, the current line status, the name of the last call created on the line, and the name of where the next incoming call will be directed.

3.3.3 Receiving notification of changes

`RLine` contains five functions that make asynchronous requests for notification of changes in the state and/or capabilities of a line; they are implemented with a corresponding function to cancel an outstanding request. Usually, an active object is used to encapsulate the asynchronous request function and handle the completion of both the request and the cancellation functions (see *Symbian OS C++ for Mobile Phones*, Chapter 17, Active Objects, Richard Harrison).

`NotifyIncomingCall()` is used to issue an asynchronous request to be notified of the next incoming call. It takes a `TName` parameter, which, when the request completes, contains the name of the incoming call as demonstrated by this code fragment:

```
class CIncomingCall: public CActive
   {
     // more code
   void IssueRequest();
   void RunL();      // from CActive
   void DoCancel();  // from CActive
     // more code
   RLine iLine;
   TName iName
```

```
};

void CIncomingCall::IssueRequest()
{
      // issue request to watch out for incoming calls
   iLine.NotifyIncomingCall(iStatus, iName);
   SetActive();
}

void CIncomingCall::RunL()
{
   if (iStatus.Int() == KErrNone)
   {
        // received notification of incoming call

        // re-issue the request (optional)
        IssueRequest();
   }
}

void CIncomingCall::DoCancel()
   {
   iLine.NotifyIncomingCallCancel();
   }
```

`NotifyHookChange()` issues an asynchronous request to be notified of a change in the hook status of the line. It takes an `RCall::THookStatus` parameter, which, when the request completes, contains the line's hook status.

`NotifyStatusChange()` issues an asynchronous request to be notified of a change in the status of the line. It takes an `RCall::TStatus` parameter, which, when the request completes, contains the line's status.

`NotifyCallAdded()` issues an asynchronous request to be notified when a new call is added to the line. It takes a `TName` as a parameter that contains the name of the call added to the line when the request completes.

`NotifyCapsChange()` issues an asynchronous request to be notified of changes to the capabilities of the line. When the request completes, the `TCaps` parameter contains the new capabilities of the line.

3.4 Call Subsession — RCall

`RCall` provides access to the functionality associated with a specific call. A call provides the basic telephony functionality, i.e., dial, answer and hang up. Information about a call's status and capabilities can be queried and notification of changes in these can be requested.

3.4.1 Overloaded functions

`RCall` has both synchronous and asynchronous methods in multiple overloaded versions; many of the asynchronous methods take a `TRequestStatus` parameter while others take a `TCallParamsPckg`.

The `TCallParamsPckg` contains specific call parameters in the form of a package buffer representing a `TCallParam`. Because some of these functions can take a long time to complete, use of an asynchronous variant of the particular function is recommended.

3.4.2 How to open a call

To open a call, either `OpenNewCall()` or `OpenExistingCall()` are used; both functions have multiple overloaded forms. Every call is given a name in the form `Phone_Name::Line_Name::Call_Name` that identifies the call uniquely with the line and phone it is opened on. `OpenNewCall()` can take an `RTelServer`, `RPhone`, or `RLine` subsession as a parameter, taking as a second parameter the name of the line to open the call on, which is not needed when opening a call with a line subsession:

```
TInt OpenNewCall(RTelServer& aServer, const TDesC& aName);

TInt OpenNewCall(RPhone& aPhone, const TDesC& aName);

TInt OpenNewCall(RLine& aLine);
```

These functions are further overloaded with forms that can retrieve the name of the call that has been opened:

```
TInt OpenNewCall(RTelServer& aServer, const TDesC& aName, TDes&
  aNewName);

TInt OpenNewCall(RLine& aLine, TDes& aNewName);

TInt OpenNewCall(RPhone& aPhone, const TDesC& aName, TDes&
  aNewName);
```

Note: A call that has been opened has to be explicitly closed with `Close()`, otherwise a memory leak will result.

3.4.3 Call ownership

Each call has an owner, and only the owning client can terminate or gain control of the call. Other clients can, however, monitor the call or view its state.

The client that initially connects to the call has ownership; however, ownership can be given up to other interested clients, e.g., one client may make the call, while another uses the call to transfer data.

A client gives up ownership by calling **TransferOwnership()**, which returns **KErrEtelNoClientInterestedInThisCall** if it fails.

A client gains ownership by using **AcquireOwnership()**, which completes when the owning client calls **TransferOwnership()**, or closes its handle to the call without hanging up. The request to gain ownership of the call can be cancelled with **AcquireOwnershipCancel()**.

3.4.4 How to make a call

To dial a number, a call has to be already opened. The call mode is decided by whether the call has been opened on a voice or data line. The function to dial a number is `Dial()`; it has four overloaded versions, each taking an `aTelNumber` parameter, which is a descriptor containing the number to dial:

```
TInt Dial(const TTelNumberC& aTelNumber) const;

TInt Dial(const TDesC8& aCallParams, const TTelNumberC&
```

```
  aTelNumber) const;

void Dial(TRequestStatus& aStatus, const TTelNumberC&
  aTelNumber);

void Dial(TRequestStatus& aStatus, const TDesC8& aCallParams,
  const TTelNumberC& aTelNumber);
```

An example code fragment using `Dial()` is given by:

```
RLine line;
  // More Code
RCall call;
call.OpenNewCall(line);
_LIT(KPhoneNumber, "01611110000")
call.Dial(KPhoneNumber); // dial synchronous
```

`DialCancel()` can be used to cancel an outstanding dialing request placed with an asynchronous form of `Dial()`.

3.4.5 How to hang up a call

There are asynchronous and synchronous forms of `HangUp()` that terminate a call on a line established using `AnswerIncomingCall()`, `Dial()`, or `Connect()`.

`HangUpCancel()` can be used to cancel a request to hang up, placed using the asynchronous form of `HangUp()`. It is unlikely once the request to terminate the call has been made that the call can be retained.

The following example shows how to hang up a call. User::WaitForRequest() is used to wait for the asynchronous call. This is done for illustrative purposes only. Normally an active object should be implemented when making an asynchronous call.

```
RCall call;
  // More code
TRequestStatus status;
call.HangUp(status);
User::WaitForRequest(status);
```

3.4.6 Querying a call properties

As with `RPhone` and `RLine`, `RCall` provides functions that give information about the state and capabilities of a call. To retrieve information about the current call, `GetInfo()` takes an `RCall::TCallInfo` parameter that contains the call name, line name, hook status, status of the call, and duration.

`GetStatus()`and `GetCallDuration()` return the call status and call duration, respectively.

Further information about the call can be retrieved using `GetCallParams()` and `GetBearerServiceInfo()`. `GetCallParams()` returns the call parameters, while `GetBearerServiceInfo()` can only be used when the call is active and returns information about the bearer capabilities and the speed of the bearer.

3.4.7 Receiving notification of changes in the call

Change notification in a call can be requested in a manner similar to the way notification of changes in the line can be requested. `NotifyHookChange()` and `NotifyStatusChange()` replicate the behavior of the identically named functions in `RLine`, while `NotifyCapsChange()` requests notification of changes in capabilities of a call. These three functions have respective cancel functions to cancel any outstanding requests.

Using `NotifyCallDurationChange()` it is possible to gain notification of changes in a call length; the request completes when the call duration has increased by one second and its `TTimeIntervalSeconds` parameter contains the current update of the call's duration.

3.4.8 How to answer a call

`AnswerIncomingCall()` provides this functionality and exists in both synchronous and asynchronous forms. Because answering a call can take some time, use of an asynchronous form is recommended. If there is an incoming call, that call will be answered, otherwise the next incoming call will be answered. Both forms have overloaded versions taking call parameters.

After an incoming call alert (which uses `RLine::NotifyIncomingCall())` has been received, the call needs to be opened using `OpenExistingCall()` before the call can be opened using `AnswerIncomingCall()`. For example:

```
RCall call;
call.OpenExistingCall(aLine, aCallName);
call.AnswerIncomingCall(iStatus);
```

`AnswerIncomingCallCancel()` cancels a request made with `AnswerIncomingCall()`. If a call has arrived, then an attempt is made to stop answering that call; otherwise, the TSY notes that it should not answer the call if it arrives.

3.4.9 Data calls

A data call has to be opened on a line that has data capability. Once connected, the client can access the port directly to transfer data using either the synchronous or asynchronous form of `LoanDataPort()` that takes a `TCommPort` parameter, which, when the request completes, contains the CSY module name and the name of the port.

When this function has completed, ETel queues any commands it has for the modem, leaving the port free for client data. When the client has finished with the port, calling `RecoverDataPort()` returns control of the port to the ETel server.

```
RCommServ commServer;
RComm comm;
RCall::TCommPort commPort;
TRequestStatus status;

call.LoanDataPort(status, commPort);
User::WaitForRequest(status);

//Tint err= commServer.Connect();
User::LeaveIfError(commServer.Connect());
```

```
// load correct CSY
User::LeaveIfError(commServer.LoadCommModule(commPort.iCsy));

User::LeaveIfError(comm.Open (commServer, commPort.iPort,
  ECommShared));
  // More Code
comm.Write(status,buf);
User::WaitForRequest(status);
  // More Code
comm.Close();
call.RecoverDataPort();
commServer.Close();
```

4. AT Commands

The Hayes AT (ATtention) commands comprise a standard set of instructions for controlling a modem. Developed in the mid 1980s by Hayes Micro Computers, the commands were introduced to control a new generation of smart modems because previously, modems had no memory and were not programmable. The Hayes AT command set is now regarded as the de facto standard for modem control.

A Hayes-compatible modem has two modes: *command* and *on-line*. Any command(s) issued while in on-line mode are treated as data except for an escape string, which returns the modem to command mode. Usually the use of AT commands is hidden behind higher-level APIs, but occasionally it is useful to access the modem directly.

Extensions to the AT standard have been made to support the additional functionality of GSM devices, such as Short Message Service (SMS), as well as additions to the command set by individual device manufacturers for their particular hardware. However, there is no reason to assume that all GSM devices or network operators support all the commands in the standards.

4.1 Basic AT Command Syntax

The AT command syntax is straightforward — all commands, with two exceptions, must begin with the characters `AT`, which alerts the modem to accept a command string. The command buffer parse ignores spaces, and content can be in upper- or lowercase, but AT command syntax itself cannot be mixed case. Therefore, while `ATDT`, `atdt`, and `ATZ\r\n` are legal AT commands, `aTDt` and `ATz\r\n` are not. If the buffer contains a syntax error, then the value `ERROR` is returned by the modem and the input is ignored. If the command is accepted, the modem usually returns the value `OK`.

Examples of AT commands are:

`ATDT 01611110000`

`AT`tention `D`ial (`T`one) the number: `01611110000`

`ATS0=1&W0&Y0`

`AT`tention `S`et (register `0`) to the value: `1&W0&Y0`

When a call is dialed or answered, the call is automatically placed in online mode. By issuing the escape sequence `+++` the modem can be returned to command mode and back to online mode by issuing the `AT0` command. It is possible to gain help on a command by adding `=?`, which returns all supported parameters.

4.2 AT Commands on the Series 60 Platform

There is no public API to send AT commands directly to a modem, however, access to the communication channel used by the modem can be gained and used to pass AT commands to it.

AT commands can always be used to communicate with the modem and are usually implemented in order to access functionality that is not available using the ETel APIs. However, due to the way the ETel telephony server is designed, it is not possible for a client

to receive any responses from the modem. This behavior limits the usefulness of utilizing AT commands because the client cannot tell if the command issued has succeeded.

There are two ways of writing AT commands to the modem. If there is an active data call, `LoanDataPort()` gives direct access to the communication channel. By issuing the AT command "`+++`" the modem is switched to command mode.

Note: It is important to note that the `+++` command requires a so-called guard time before and after sending the command. Usually the time is between one and two seconds.

When writing an AT command to a modem, the escape sequence `\r\n` has to be appended to the command string, as shown:

```
call.LoanDataPort(status, commPort);

  // More code (including guard time)

_LIT8(buf8, "+++");
comm.Write(status, buf8);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());

  // More code (including guard time)

_LIT8(hangup8, "ATH\r\n");
comm.Write(status, hangup8);
User::WaitForRequest(status);

  // more code
```

The alternative method is to open a communications channel to the modem directly, as demonstrated in this fragment:

```
RCommServ commServer;
RComm comm;
commServer.Connect();

_LIT(KCsyName, "dataport");
User::LeaveIfError(commServer.LoadCommModule(KCsyName));

_LIT8(KDataPort, "DATAPORT::0");
TBufC<12> port = KDataPort;
TInt ret= comm.Open(commServer, port, ECommShared);

  // More code

_LIT8(buf8, "ATZ\r\n");    // return modem to its default state
comm.Write(status, buf8);
User::WaitForRequest(status);

  // More code
```

Caution: Using AT commands directly rather than through ETel APIs can be dangerous and cause the system to fail or crash unpredictably.

4.3 Summary of AT Commands

It is important to realize that use of AT commands is limited by the restriction of being unable to receive responses from the modem. This said, the ability to issue AT commands directly can be advantageous in some circumstances, particularly if attempting control of vendor-specific or custom hardware. Further examples of AT commands appear below:

`ATD`{Dial modifier}{phone number}

Dials the given number using the requested dial modifiers (if any), e.g., `ATD0161111000`, `ATD;01611110000`, and resumes command mode after dialing.

`ATA`

Answers the incoming call.

`ATH`

Hangs up.

`ATO`

Returns to online mode; normally used after “`+++`” has been used to enter command mode.

Extensions to the standard AT command set enable the issue of other types of modem commands such as GSM in this manner. GSM commands have the general, though not exclusive format `AT+Cxxx`. For example:

`AT+CHUP`

hangs up, or

`AT+CPBS={storage}`

sets the phone book storage mechanism to use, e.g., `AT+CPBS=SM` sets the phone book to the SIM card. Other common storage options for the `CPBS` subcommand are:

`FD` SIM fix-dialing phone book

`ME` ME phone book

`DC` ME dialed calls list

`ON` SIM (or ME) own numbers (Mobile Subscriber Integrated Services Digital Network — MSISDN) list

`LD` SIM last-dialing phone book

`MC` ME missed (unanswered received) calls list

`RC` ME received calls list

Any AT command understood by the target modem can be sent by this mechanism and handled in this way.

5. Summary

The ETel APIs implement a powerful and extensible management interface to the telephony features of any Series 60 device. By implementing only core telephony for itself and an extensions API to handle specific hardware abilities via TSY device drivers, the hierarchy supplied by ETel allows developers considerable flexibility in handling and controlling telephony features on different hardware platforms without compromising compatibility across hardware and network service providers. The implementation allows future generations of hardware and network services to be implemented with relative ease.

ETel support for sending commands directly to the modem via the AT command set further enhances this flexibility.

6. Terms and Abbreviations

Term or Abbreviation	Meaning
API	Application Programming Interface
AT	Hayes ATtention Command/Command Set
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
TSY	ETel Telephony Extension Module