McGraw-Hill TELECOM
PROFESSIONAL

MOBILE APPLICATION DEVELOPMENT

with SMS and the SIM Toolkit

Building Smart Phone Applications

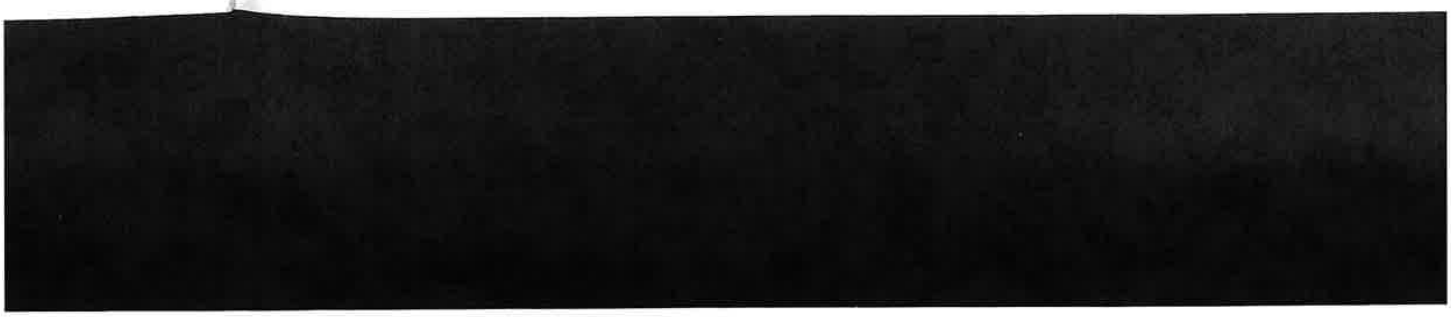
SCOTT B. GUTHERY • MARY J. CRONIN

Mobile Application Development
with SMS and the SIM Toolkit

Scott B. Guthery

Mary J. Cronin

McGraw-Hill
New York • Chicago • San Francisco • Lisbon
London • Madrid • Mexico City • Milan • New Delhi
San Juan • Seoul • Singapore • Sydney • Toronto



McGraw-Hill

A Division of The McGraw-Hill Companies

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9 8 7 6 5 4 3 2 1

ISBN 0-07-137540-6

The sponsoring editor for this book was Marjorie Spencer, the editing supervisor was Steven Melvin, and the production supervisor was Sherri Souffrance. It was set in Vendome by Patricia Wallenburg.

Printed and bound by R. R. Donnelley & Sons Company.

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, Professional Publishing, McGraw-Hill, Two Penn Plaza, New York, NY 10121-2298. Or contact your local bookstore.

CHAPTER 2

Basic SMS Messaging

There are many software development kits and products on the market that you can use to connect your application to SMS messaging. These range is from very low-level packages that simply connect a serial line port to the mobile phone up to all-singing, all-dancing packages that provide all sorts of message management services. In between are packages that provide various APIs to SMS messaging such as Telephone Aplication Program Interface (TAPI) that make it easy to integrate SMS messaging into existing application suites.

We will begin with basic, low-level messaging and work our way up the food chain. You may never actually build an application using these low-level commands but it's good to know what's under the hood and what's possible just in case you get stuck and have to reach for the spanners. The higher-level packages are essentially fancy ways of generating those low-level commands.

In the next couple of paragraphs we discuss setting up your mobile application development workbench.

Connecting the Handset

Every GSM and 3GPP handset is an air interface modem and a plain old telephone handset. This means you can connect the handset to an external interface on your computer and send it AT commands just as you did with your dial-up modem. The physical connection can be any one that your computer offers such as a serial port, a USB port, or an IrDA port. We are going to use a serial COM port for the examples in this chapter because it is the most widely used one at present.

Besides an activated GSM phone you'll need a cable that connects the phone and the serial port on your computer. You'll also need to install a modem driver on your computer that knows how to talk to the phone. The cable and the driver depend on the model of the handset you are using. Most handset manufacturers offer a data kit of some sort for their handsets that includes the right cable and the driver. Examples in this chapter use a Nokia 5190 handset and the SoftRadius driver and cable for that handset from Option Inc. Nokia produces several very nice data kits called the Nokia Data Suite and the Nokia PC Connectivity SDK, which accomplish the same thing.

After you've installed the driver, you can use the same terminal program that you use for dial-up modems to test the connection. On a Windows system, just use HyperTerminal. Type "AT" on the COM

port connected to the phone. If everything is working properly, you should see "OK." Now you're ready to start building SMS applications.

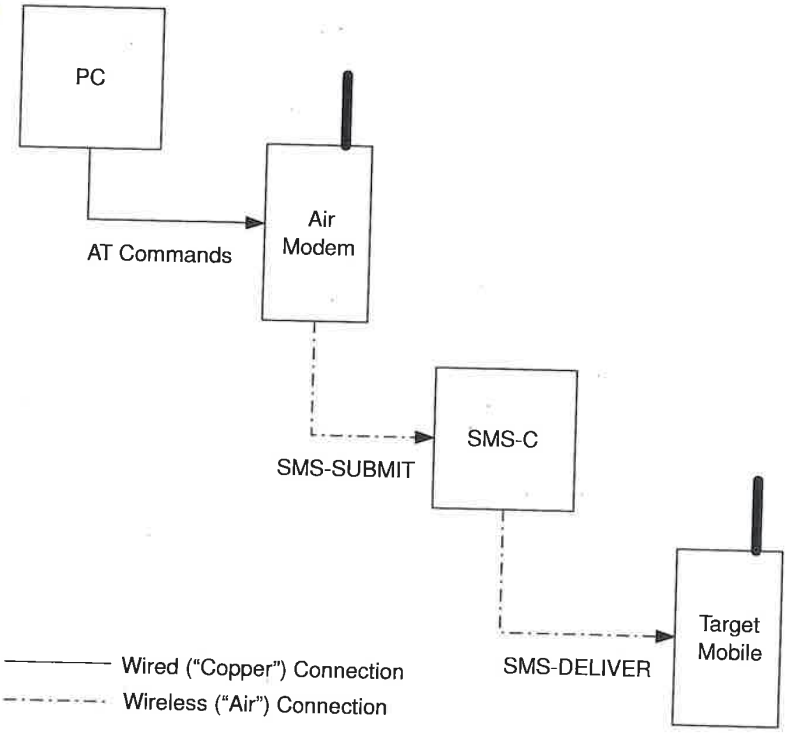


Figure 2-1
Message flow from desktop PC to mobile handset.

Communicating with the Handset

In addition to many of the standard V.32ter and Hayes modem dial-up modem AT commands, your mobile handset supports a set of AT commands that are particular to connecting to the GSM network and sending short messages. If you're a gnarly old Hayes modem hacker, you'll feel right at home. The standard handset AT commands are described in the following two documents:

- **3GPP 27.005**—DTE-DCE interface for SMS and CBS
- **3GPP 27.007**—AT command set for 3G UE

The big difference between using a dial-up modem connected to the landline telephone network and a handset connected to the GSM network is how much you can see and say to the network itself. About the only thing you said to the network through your dial-up modem was "Connect me to the following number." You did this with the Hayes ATDT command.

```
ATDT 6172345678
```

This caused the dial-up modem to generate the right dual tone mulit-frequency (DTMF) tones on the line to cause the telephone network to set up a dedicated circuit connection between your modem and the modem that answered at the other end. Once the connection was established, all the wired network did was move an analog signal from one end to the other. The modems on both ends took care of turning the analog signal into bits, frames, packets, and messages.

A mobile network is continuously and more intimately involved in the bit stream if for no other reason than the modem you are trying to communicate with—the mobile handset out there somewhere—keeps moving around.

In the 27.007 AT command set you will find some old friends such as ...

ATD	Dial command
ATE	Command echo
ATH	Hang up call
ATA	Answer call
ATS	Select an S-register
ATQ	Result code surpression
ATZ	Recall stored profile

But you'll also find lots of commands that are more about you talking to and about the network than to and about the handset modem such as ...

AT+CSCS	Select character set
AT+WS46	Select wireless network
AT+CBST	Select bearer service type

AT+CRLP	Radio link protocol
AT+CR	Service reporting protocol
AT+CRC	Cellular result codes
AT+COPS	Operator selection
AT+CSCA	Service center address

Finally, because a mobile handset is a much more capable device than the old V.32 Hayes modem, there are many commands that you can use to manipulate it such as ...

AT+CPBF	Find phone book entries
AT+CPBR	Read phone book entry
AT+CPBW	Write phone book entry
AT+CMGL	List messages
AT+CMGR	Read messages
AT+CMGS	Send message

For example, after I connected my mobile phone to my PC and fired up HyperTerminal, I used AT + CMGL to get a list of the messages that were stored in the SIM:

```
AT
OK
AT+CMGL
+CMGL: 1,1,24
07919171095710F0040B917118530400F900001030804065535805C8
  329BFD06
+CMGL: 2,1,30
07919171095710F0040B917118530400F900001030111041805800CC8
  329BFD6681EE6F399B0C
+CMGL: 3,1,23
07919171095710F0040B917118530400F900001030111061255804E5
  B2BC0C
+CMGL: 4,1,25
07919171095710F0040B917118530400F900001030111002035806650
  79595E9603
+CMGL: 5,1,24
```

```
07919171095710F0040B917118530400F900001030111064545805C8
  329BFD06
+CMGL: 6,1,28
07912160130300F4040B917118530400F900001080507092446 90AD4
  F29C0E8A8164A019
+CMGL: 7,1,37
07912160130300F4040B917118530400F900001080507003516914D7
  329BCD02A1CB6CF61B947FD7
E5F332DB0C

OK
```

Figure 2-2 SMS message headers.

There were seven messages stored in the SIM. In Chapter 3, we will analyze the numbers and find not only the message but also lots of interesting information about the message such as who sent it and when it arrived.

Communicating with the Network

Because the mobile network is an active participant in moving messages between your application and a mobile device, you have to be much more concerned with the details of formatting the messages you send. Remember the mobile network actually looks at the bytes in your message (actually in the headers on your message) to figure out what to do with it. "Please tell Sally Green wherever she is that dinner won't be ready until 7" just doesn't hack it.

We will discover that there are lots of things besides who should receive the message that you can tell the GSM network and its SMS centers (SMSC). The string of bytes that you send into the network contains not only the message but also lots of other information that instructs the network as to how and when you want this to happen.

The two standards that govern the construction of SMSs what we will be using are:

- **3GPP 23.040**—Technical realization of SMS
- **3GPP 24.011**—PP SMS support on the mobile radio interface

These standards cover the encoding of the message that gets delivered to the destination handset and the encoding of the instructions to the GSM network and the SMSC.

11064545805C8

0709244690AD4

07003516914D7

Remember our discussion in Chapter 1 about the encapsulation of protocols? In building low-level commands for sending SMSs, we are in fact talking to three separate entities: the local handset to which we are sending AT commands, the network and its SMSC, and the end-point mobile that will receive the message.

Figure 2-2 shows the complete SMS header diagram. We are covering only the outermost two in this chapter and will get to the others in later chapters. You build all the headers, so you will have to remember whom you are talking to and what you are saying to them as you build your SMS message.

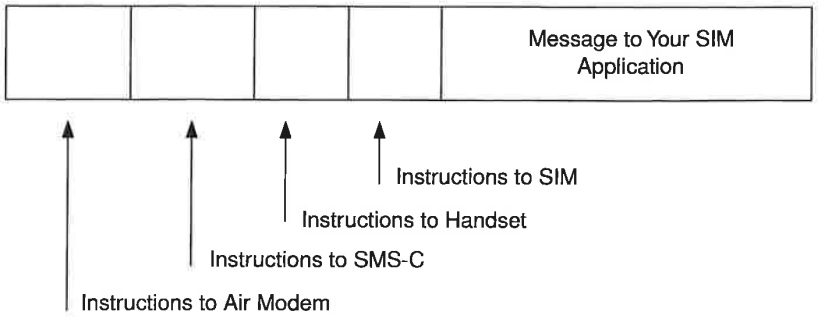


Figure 2-2 SMS message headers.

Hello, Mobile World

Let's start by opening a serial port connection to the local handset. My Nokia 5190 is connected to COM5, so using the C programming language I'd write:

```
handle = CreateFile("COM5",
      GENERIC_READ | GENERIC_WRITE, // read and write
      0, // exclusive access
      NULL, // no security
      OPEN_EXISTING,
      0, // no overlapped I/O
      NULL); // null template
```

It is on this connection that we will send AT commands to the local handset that in turn will relay the information to the GSM network.

We must set this serial connection to binary so that the operating system and its drivers don't touch the data as it passes through, for example, by adding carriage returns and line feeds. We want the data we construct to get to the handset and to the network exactly as we built it and not with any "help" from folks along the way.

How this is done changes from handset driver to handset driver. For the particular driver I'm using, binary information sent on this connection is hex-encoded as ASCII characters, so if you wanted to send the byte 0x9D, you'd send the ASCII string "39 44": 39 is the hexadecimal value for the ASCII character 9 and 44 is the hexadecimal value for the ASCII character D.

Let's start by sending a simple "Hello, world" message to the mobile phone at +1 617-230-1346.

What we do is pack in a hex-encoded byte blob all the information needed to get this message to its destination along with the message itself and ship this blob off to the carrier's SMSC which in turn will get it to where it is going.

The byte blob is an SMS_SUBMIT Transfer Protocol Data Unit (TPDU). We'll take a detailed look at TPDUs in Chapter 3. The one at hand consists of the following fields:

1. Transfer protocol parameters = 0x01 (an SMS_SUBMIT TPDU)
2. Message reference number = 0x00 (let the handset assign it)
3. Length of destination number in digits = 0x0B (11 digits)
4. Type of destination number = 0x91 (international format)
5. Destination telephone number (nibble swapped) = 0x6171321043F6
6. Protocol identifier = 0x00 (implicit)
7. Data coding scheme = 0x00 (GSM default alphabet)
8. Message length = 0x0C (there are 12 characters in "Hello, world")
9. Message = 0xC8329BFD6681EE6F399B0C ("Hello, world")

The coding of the actual message, "Hello, world," requires some explanation. No stone is left unturned when it comes to optimizing the use of the air interface. If we had transmitted the ASCII characters as bytes, we would have wasted a bit for every character we sent because ASCII characters are coded on 7 bits and sending this message as an 8-bit byte wastes 1 bit. Now, 1 bit is no big deal if you have megabytes of memory and gigabytes of disk space, but on an air interface this represents a waste of one-eighth of the channel capacity and this cannot be tolerated.

What we do is very simple. First, we put the first character into the first byte. Next, we take the low-order bit of the seven bits of the second ASCII character and stuff it in the unused high-order bit of the first byte. Now we put the six remaining bits of the second character into the second byte. Next, we take the two low-order bits of the seven bits of the third ASCII character and stick them into the two unused high-order bits in the second byte, and so forth.

Here's the result of applying this packing algorithm to "Hello, world":

	H	e	l	l	o	,		w	o	r	l	d
Unpacked	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64
Packed	C8	32	9B	FD	66	81	EE	6F	39	9B	0C	

In this case, the low-order bit of the ASCII character "e" is 1, so we put that into the high-order bit of the first byte, which is unused after we put the seven bits of "H" into the low-order bits of the byte. This turns 0x48 into 0xC8. Now, we take the remaining low-order six bits of "e," 0x25, and put them into the low-order six bits of the second byte. We then put the two low-order bits of the seven bits of the ASCII character "l," namely 0 and 0, into the two unused high-order bits of the second byte. This yields a second byte of 0x32, and so forth. As a result, we save a complete byte.

Here is some C code that performs this packing and also makes the hex-encoded ASCII characters that are sent to the handset:

```
#include <string.h>

void unpack78(char *p, int n, char *s);
void pack78(char *s, char *p, int *n);

#define N(c) (c<=0x39?((c)-0x30):((c)-0x37))
#define M(c) (c<=0x09?((c)+0x30):((c)+0x37))

void pack78(char *s, char *p, int *n)
{
  unsigned char byte[160];
  int bits, i, j, k;

  k = strlen(s);
```

```
  /* Pack the ASCII characters into bytes */
  for(i = j = bits = 0; i < k; i++, j++) {
    if(bits == 7) {
      bits = 0;
      i++;
    }
    byte[j] = (s[i]&0x7F)>>bits | s[i+1]<<(7-bits);
    bits = (++bits)%8;
  }

  /* Convert bytes to ASCII nibbles */
  for(i = 0; i < j; i++) {
    k = (byte[i]>>4)&0x0F;
    *p++ = M(k);
    k = byte[i]&0x0F;
    *p++ = M(k);
  }

  *n = j;
}
```

For the sake of completeness, here is the corresponding unpacking routine that we will need when we receive messages from the handset:

```
void unpack78(char *p, int n, char *s)
{
  int bits, i, j;
  unsigned c, byte[160];

  /* Convert ASCII nibbles to bytes */
  for(i = 0; i < n; i++) {
    byte[i] = N(*p);p++;
    byte[i] = (byte[i]<<4)|(N(*p));p++;
  }

  /* Extract the ASCII characters from the bytes */
  for(i = j = bits = 0; j < n; i++, j++) {
    if(bits > 0)
      c = byte[i-1]>>(8-bits);
    else
```

```
        c = 0;
      if(bits < 7)
        c |= byte[i]<<bits;
      *s++ = c&0x7F;
      bits = (++bits)%8;
      if(bits == 0)
        i -= 1;
    }
    *s = '\0';
}
```

So the complete SMS_SUBMIT TPDU for "Hello, world" looks like this:

```
01000B916171321043F600000CC8329BFD6681EE6F399B0C
```

All we have to do now is use an AT command to send this TPDU off to the SMSC. This is the send-message AT command:

```
AT+CMGS=<TPDU length><CR><SMSC address><TPDU><CTRL-Z>
```

The SMSC address is the telephone number of the SMSC to which the handset should send the TPDU. Like the destination telephone number, the telephone number of the SMSC consists of three subfields:

1. Length of the telephone number in octets = 0x07 (7 octets)
2. Format of the SMS telephone number = 0x91 (international format)
3. Telephone number of SMSC (nibble swapped) = 0x9171095710F0

This particular SMSC is in the VoiceStream network, where the handset set I am using as my air modem is registered. When you test this example, you'll have to replace this phone number with the phone number of the SMSC in the network to which you subscribe.

The following code is what I write to COM5 to send "Hello, world" to my mobile:

```
AT+CMGS=24
07919171095710F001000B916171321043F600000CC8329BFD6681EE
  6F399B0C^Z
```

The <CR> is the byte 0x0D and the CTRL-Z is the byte 0x1A in the actual sequence of bytes sent to the handset. Give it a try. I look forward to receiving your SMS!

So what happens if the mobile we send a message to sends one back? We can list all the messages in the handset using the AT command:

```
AT+CMGL
```

and then we can retrieve the one we want by using the AT command:

```
AT+CMGR=<index>
```

When using AT+CMGR to retrieve the latest message that arrived, the handset replies with:

```
+CMGR: 1,21
07919171095710F0 040B916171321043F6 00 00 10201180234458
  02C834
```

The 1 on the first line indicates the status of the message. The number 1 means this is a received message that has been read. The following 21 is the number of bytes in the TPDU in the following data. The following line shows the data comprising the message. It is in the same general format as the data in the AT+CMGS command, namely the SMSC telephone number followed by a TPDU. In this case, however, it is the telephone number of the SMSC delivering the message and an SMS_DELIVER TPDU rather than an SMS_SUBMIT TPDU. In other words, the TPDU is being delivered to the handset rather than the handset submitting a TPDU to the network. We'll discover that what is delivered is not exactly the same as what is submitted.

The SMSC phone number is just like the one we sent to, so let's analyze the SMS_DELIVER TPDU. We'll be using 3GPP TS 23.040 to do this.

1. Transfer protocol parameters = 0x04 (SMS_DELIVER with no more coming)
2. Length of originating address = 0x0B (11 bytes)
3. Type of originating address = 0x91 (international format)
4. Originating address (nibble swapped) = 0x6171321043F6 (+1 617 230 1346)

5. Protocol identifier = 0x00
6. Data coding scheme = 0x00
7. Service center timestamp (nibble swapped) = 0x10 20 11 80 23 44 58 (Y/M/D/H/M/S/Zone = 2001 February 11, 8:32:44, GMT-5)
8. Length of message = 0x02 (2 bytes)
9. Message = 0xC834 ("Hi")

To unpack 0xC834, just take the top bit from 0xC8 and put it to the right of 0x34. This turns 0xC8 into 48 and 0x34 into 0x69.

	H	i
Packed	C8	34
Unpacked	48	69

Therefore, at 8:32 Eastern Standard Time the mobile phone +1 617 230 1346 sent me the message "Hi." I'm a fascinating conversationalist when I'm talking to myself.

You can achieve a variety of special effects by setting various parameters in the SMS_SUBMIT TPDU. We will cover TPDUs in detail in Chapter 3, but to give you a feel for what is possible, suppose we'd like to have our message pop up on the screen and at the same time provide a way to quickly open up a telephone call back to the sender. This is useful for sending alerts needing an immediate response. All the recipient has to do is pick "Use Number" or "Return Call" or whatever phrase his or her handset uses to indicate the presence of a return call number in the SMS.

The pop up message is accomplished by setting the Data Coding Scheme byte to 0xF0 rather than to 0x00. This says "put the message on the screen don't just store it in the SIM" and tells the user that a new message has arrived. Creating a return call path is accomplished by setting the Protocol Identifier to 0x5F rather than to 0x00. Give it a try.

You can see why we call SMS messaging the assembly language programming of the wireless network. An SMS header is in a very real sense an instruction for a very large instruction word (VLIW) computer where the computer is the mobile telephone network. Every bit counts and these bits interact. Further, the network executes many of the fields of the instruction in parallel as if it were horizontal microcode. If you missed the era of bit-slice computers, now's your chance.

There are many other AT commands that you can send to the handset. We can't go into all of them here. You can download 3GPP TS 27.005 and 3GPP TS 27.007 for the complete story. All phones that support data support the major sending and receiving AT commands. You'll have to experiment to find out which of the more esoteric commands such as AT+CUSD (unstructured supplementary service data) and AT+CMER (mobile equipment event reporting) are supported with your driver and phone combination.

Table 2-1 lists some of the error codes you might run into when an AT command returns an error.

TABLE 2-1
SMS Error Codes

Error	Error Meaning
0-127	GSM 04.11 Annex E-2 values
128-255	GSM 03.40 section 9.2.3.22 values
300	Phone failure
301	SMS service not available
302	Operation not allowed
303	Operation not supported
304	Invalid PDU mode parameter
305	Invalid text mode parameter
310	SIM not inserted
311	SIM PIN needed
312	SIM PIN2 needed
313	SIM failure
314	SIM busy
315	SIM incorrect
320	Memory failure
322	Memory full
331	No network
332	Network timeout
500	Unknown error
512	Manufacturer specific error

Table 2-2 gives the GSM 7-bit default alphabet as specified by 3GPP 23.038 and the corresponding ISO-8859 decimal codes where applicable.

TABLE 2-2 SMS 7-Bit Default Character Encoding

Hexameter	Decimal	Character name	Character	ISO-8859 Decimal
0x00	0	Commercial AT sign	@	64
0x01	1	British monetary unit—pound	£	163
0x02	2	US monetary unit—dollar	$	36
0x03	3	Japanese monetary unit—yen	¥	165
0x04	4	Lowercase e with accent grave	è	232
0x05	5	Lowercase e with accent acute	é	233
0x06	6	Lowercase u with accent acute	ú	250
0x07	7	Lowercase i with accent grave	ì	236
0x08	8	Lowercase o with accent grave	ò	242
0x09	9	Uppercase C with cedilla	Ç	199
0x0A	10	Line feed (\n)		10
0x0B	11	Uppercase O with stroke	Ø	216
0x0C	12	Lowercase o with stroke	ø	248
0x0D	13	Carriage return (\r)		13
0x0E	14	Uppercase A with ring	Å	197
0x0F	15	Lowercase a with ring	å	229
0x10	16	Uppercase Greek delta	Δ	
0x11	17	Underline character	_	95
0x12	18	Uppercase Greek phi	Φ	
0x13	19	Uppercase Greek gamma	Γ	
0x14	20	Uppercase Greek lambda	Λ	
0x15	21	Uppercase Greek omega	Ω	
0x16	22	Uppercase Greek pi	Π	
0x17	23	Uppercase Greek psi	Ψ	
0x18	24	Uppercase Greek sigma	Σ	

continued on next page

TABLE 2-2 SMS 7-Bit Default Character Encoding (continued)

Hexameter	Decimal	Character name	Character	ISO-8859 Decimal
0x19	25	Uppercase Greek theta	Θ	
0x1A	26	Uppercase Greek xi	Ξ	
0x1B	27	Escape to extension table		
0x1B0A	27 10	Form feed (\f)		12
0x1B14	27 20	Circumflex	^	94
0x1B28	27 40	Left curly bracket	{	123
0x1B29	27 41	Right curly bracket	}	125
0x1B2F	27 47	Backslash	\	92
0x1B3C	27 60	Left square bracket	[	91
0x1B3D	27 61	Tilde	~	126
0x1B3E	27 62	Right square bracket	]	93
0x1B40	27 64	Vertical stroke	|	124
0x1B65	27 101	Euro sign		164
0x1C	28	Uppercase AE	Æ	198
0x1D	29	Lowercase ae	æ	230
0x1E	30	Lowercase German ss	ß	223
0x1F	31	Uppercase E with circumflex	Ê	202
0x20	32	Space		32
0x21	33	Exclamation mark	!	33
0x22	34	Question mark	?	34
0x23	35	Hash sign	#	35
0x24	36	General currency sign	¤	164
0x25	37	Percent sign	%	37
0x26	38	Ampersand	&	38
0x27	39	Apostrophe	'	39
0x28	40	Left parenthesis	(	40
0x29	41	Right parenthesis	)	41
0x2A	42	Asterisk	*	42

continued on next page

TABLE 2-2 SMS 7-Bit Default Character Encoding (continued)

Hexameter	Decimal	Character name	Character	ISO-8859 Decimal
0x2B	43	Plus sign	+	43
0x2C	44	Comma	,	44
0x2D	45	Hyphen and minus sign	-	45
0x2E	46	Period	.	46
0x2F	47	Slash	/	47
0x30	48	Digit 0	0	48
0x31	49	Digit 1	1	49
0x32	50	Digit 2	2	50
0x33	51	Digit 3	3	51
0x34	52	Digit 4	4	52
0x35	53	Digit 5	5	53
0x36	54	Digit 6	6	54
0x37	55	Digit 7	7	55
0x38	56	Digit 8	8	56
0x39	57	Digit 9	9	57
0x3A	58	Colon	:	58
0x3B	59	Semicolon	;	59
0x3C	60	Less-than sign	<	60
0x3D	61	Equal sign	=	61
0x3E	62	Greater-than sign	>	62
0x3F	63	Question mark	?	63
0x40	64	Inverted exclamation mark	¡	161
0x41	65	Uppercase A	A	65
0x42	66	Uppercase B	B	66
0x43	67	Uppercase C	C	67
0x44	68	Uppercase D	D	68
0x45	69	Uppercase E	E	69
0x46	70	Uppercase F	F	70

continued on next page

TABLE 2-2 SMS 7-Bit Default Character Encoding (continued)

Hexameter	Decimal	Character name	Character	ISO-8859 Decimal
0x47	71	Uppercase G	G	71
0x48	72	Uppercase H	H	72
0x49	73	Uppercase I	I	73
0x4A	74	Uppercase J	J	74
0x4B	75	Uppercase K	K	75
0x4C	76	Uppercase L	L	76
0x4D	77	Uppercase M	M	77
0x4E	78	Uppercase N	N	78
0x4F	79	Uppercase O	O	79
0x50	80	Uppercase P	P	80
0x51	81	Uppercase Q	Q	81
0x52	82	Uppercase R	R	82
0x53	83	Uppercase S	S	83
0x54	84	Uppercase T	T	84
0x55	85	Uppercase U	U	85
0x56	86	Uppercase V	V	86
0x57	87	Uppercase W	W	87
0x58	88	Uppercase X	X	88
0x59	89	Uppercase Y	Y	89
0x5A	90	Uppercase A	Z	90
0x5B	91	Uppercase A with dieresis	Ä	196
0x5C	92	Uppercase O with dieresis	Ö	214
0x5D	93	Uppercase N with tilde	Ñ	209
0x5E	94	Uppercase U with dieresis	Ü	220
0x5F	95	Section sign	§	167
0x60	96	Inverted question mark	¿	191
0x61	97	Lowercase a	a	97
0x62	98	Lowercase b	b	98

continued on next page

TABLE 2-2 SMS 7-Bit Default Character Encoding (continued)

Hexameter	Decimal	Character name	Character	ISO-8859 Decimal
0x63	99	Lowercase c	c	99
0x64	100	Lowercase d	d	100
0x65	101	Lowercase e	e	101
0x66	102	Lowercase f	f	102
0x67	103	Lowercase g	g	103
0x68	104	Lowercase h	h	104
0x69	105	Lowercase i	i	105
0x6A	106	Lowercase j	j	106
0x6B	107	Lowercase k	k	107
0x6C	108	Lowercase l	l	108
0x6D	109	Lowercase m	m	109
0x6E	110	Lowercase n	n	110
0x6F	111	Lowercase o	o	111
0x70	112	Lowercase p	p	112
0x71	113	Lowercase q	q	113
0x72	114	Lowercase r	r	114
0x73	115	Lowercase s	s	115
0x74	116	Lowercase t	t	116
0x75	117	Lowercase u	u	117
0x76	118	Lowercase v	v	118
0x77	119	Lowercase w	w	119
0x78	120	Lowercase x	x	120
0x79	121	Lowercase y	y	121
0x7A	122	Lowercase z	z	122
0x7B	123	Lowercase a with dieresis	ä	228
0x7C	124	Lowercase o with dieresis	ö	246
0x7D	125	Lowercase n with tilde	ñ	241
0x7E	126	Lowercase u with dieresis	ü	252
0x7F	127	Lowercase a with grave	à	224

Summary

In this chapter we covered the low-level programming of SMS messages. We sent and received messages using AT commands to communicate with a GSM handset. We built a simple outgoing SMS message and we interpreted a simple incoming SMS message at the bit and byte levels using 3GPP standards. In the next chapter, we move beyond simple messages and explore the range of possibilities that are available with SMS message encoding.