## Contents

ACRONYMS AND TERMS ...................................................................................................... 1
DEFINITIONS .................................................................................................................. 2
1. ABOUT THIS DOCUMENT .......................................................................................... 3
2. INTRODUCTION ........................................................................................................ 4
3. THE M2M SYSTEM PROTOCOL 2 SOFTWARE DESCRIPTION .................................. 5
   3.1 SOCKET INTERFACE ............................................................................................ 5
   3.2 NETWORK LINK LAYER ....................................................................................... 5
   3.3 DATA LINK LAYER ............................................................................................... 5
   3.4 PLATFORM WRAPPERS ....................................................................................... 6
4. INTEGRATING THE M2M SYSTEM PROTOCOL 2 WITH APPLICATION SOFTWARE .... 7
   4.1 SOFTWARE IMPLEMENTATION ............................................................................. 7
   4.2 BUILDING THE SOFTWARE .................................................................................. 8
   4.3 PORTING THE SOFTWARE ................................................................................... 8
5. THE M2M SYSTEM PROTOCOL 2 SOCKET INTERFACE .......................................... 10
   5.1 SOCKET API ....................................................................................................... 10
      5.1.1 Creating a socket (sp2_socket) ..................................................................... 10
      5.1.2 Binding a socket (sp2_bind) ......................................................................... 11
      5.1.3 Listening to a socket (sp2_listen) ................................................................. 12
      5.1.4 Accepting a socket (sp2_accept) ................................................................... 14
      5.1.5 Connecting a socket (sp2_connect) ............................................................... 15
      5.1.6 Sending data to a socket (sp2_send / sp2_sendto) ........................................ 16
      5.1.7 Receiving data from a socket (sp2_recv) ...................................................... 18
      5.1.8 Closing a socket (sp2_close) ....................................................................... 19
      5.1.9 Resolving host information (sp2_gethostbyname) ........................................ 20
      5.1.10 Network utility functions .......................................................................... 21
      5.1.11 Handling socket descriptors (sp2_FD_ZERO, sp2_FD_SET, sp2_FD_CLR, sp2_FD_ISSET). .................................................................................................................. 21
      5.1.12 Reading an asynchronous socket (sp2_select) ........................................... 22
      5.1.13 Controlling socket mode (sp2_ioctl) ............................................................ 24
      5.1.14 Reading the latest socket status (sp2_socket_status) .................................. 25
   5.2 M2M SYSTEM PROTOCOL 2 CONTROL API ..................................................... 26
Legal Notice
Copyright © 2004 Nokia. 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.
# ACRONYMS AND TERMS

<table>
<thead>
<tr>
<th>Acronym/term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Application Module</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CHAP</td>
<td>Challenge Handshake Authentication Protocol</td>
</tr>
<tr>
<td>DLL</td>
<td>Data Link Layer</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name Service</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>KB</td>
<td>Kilobyte (equivalent to 1024 bytes)</td>
</tr>
<tr>
<td>M2M</td>
<td>Machine-to-Machine (Man-to-Machine, Machine-to-Man)</td>
</tr>
<tr>
<td>NLL</td>
<td>Network Link Layer</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>Platform</td>
<td>Operating system and communication services</td>
</tr>
<tr>
<td>PPP</td>
<td>Point-to-Point Protocol</td>
</tr>
</tbody>
</table>
# DEFINITIONS

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket API</td>
<td>M2M System Protocol 2 API for socket operations</td>
</tr>
<tr>
<td>M2M System Protocol 2 Control API</td>
<td>M2M System Protocol 2 API for serial link control</td>
</tr>
<tr>
<td>Wireless Link Control API</td>
<td>M2M System Protocol 2 API for GSM link control</td>
</tr>
<tr>
<td>Software wrapper</td>
<td>Used between software elements to isolate platform-specific features</td>
</tr>
</tbody>
</table>
1. ABOUT THIS DOCUMENT

This document describes, from application software point of view, how to integrate and use the M2M System Protocol 2. In addition to providing an overview of the M2M System Protocol 2 software, the document includes instructions on how to integrate the M2M System Protocol 2 with application software and how to use the APIs of the M2M System Protocol 2 socket interface. The document also includes an example of the M2M System Protocol 2 socket usage.
2. INTRODUCTION

The M2M System Protocol 2 software provides a socket-programming interface for application software. The socket interface may be used

- locally between Application Module (AM) applications
- to connect an AM application to a Java™ IMlet running on the Nokia M2M module
- to connect an AM application to machines on the Internet or intranet via wireless connectivity over the GSM network. The socket interface can also be used to connect an AM application to other remote machines that use IP.

Because the IP protocol stack placed at the disposal of the socket interface is located on the Nokia M2M module, application software must have a connection to the Nokia M2M module services. As illustrated in Figure 1, the M2M System Protocol 2 software is used to tunnel IP traffic between the AM and the Nokia M2M module via a serial connection.

Figure 1. M2M System Protocol 2 overview

The socket interface provides a standard interface towards application software. In addition to socket handling, the socket interface makes it possible for the application to control both the wireless and serial connection. The M2M System Protocol 2 software connects the AM to the Nokia M2M module via a serial connection link, and enables the control of a M2M wireless connectivity link.

The serial connection is a physical connection between the AM and Nokia M2M module. It is connected between a serial port of the AM and the M2M System Connector of the Nokia M2M module. The serial connection defines the lowest layer transmission format using a point-to-point type of serial mode transmission as described in the Nokia M2M System Protocol 2 Specification.

The wireless connectivity makes it possible to connect the Nokia M2M module to a remote peer over a GSM network. Thus application software can be connected to sockets located on remote machines.
3. THE M2M SYSTEM PROTOCOL 2 SOFTWARE DESCRIPTION

The M2M System Protocol 2 is based on the Open Systems Interconnection (OSI) Reference Model. It is a set of layers organized in a hierarchical structure, where each layer uses the layer below and provides services for the layer above. When data is sent from application software to a network, it flows downward the stack layer by layer until it has passed through the whole stack. When the data reaches the bottom layer it is passed to a network. The data received from the network flows the stack in an opposite direction. The software structure of the M2M System Protocol 2 is illustrated in Figure 2.

![Figure 2. M2M System Protocol 2 software structure](image)

3.1 SOCKET INTERFACE

The socket interface is on the top level of the M2M System Protocol 2. It provides all the services that application software uses for socket communication over the network. For more information on the socket interface, see Chapter 5.

3.2 NETWORK LINK LAYER

The Network Link Layer (NLL) is used to carry the M2M System Protocol 2 protocol primitives (requests/responses and indications/notifications) between the AM and the Nokia M2M module. For more information on the NLL, see the *Nokia M2M System Protocol 2 Specification*.

3.3 DATA LINK LAYER

The Data Link Layer (DLL) handles the transmission of protocol data packets between the AM and the Nokia M2M module over a serial connection. For more information on the DLL, see the *Nokia M2M System Protocol 2 Specification*. 
3.4 PLATFORM WRAPPERS

The M2M System Protocol 2 is designed to operate in heterogeneous environments where a variety of operating systems and different types of serial communication hardware are used. The operating system and serial communication wrappers separate platform-specific functionality from the protocol stack in order to enhance the portability of the M2M System Protocol 2 implementation to different environments. For more information on the M2M System Protocol 2 platform wrapping, see Chapter 4.
4. INTEGRATING THE M2M SYSTEM PROTOCOL 2 WITH APPLICATION SOFTWARE

This chapter describes how the M2M System Protocol 2 is integrated as a part of application software.

4.1 SOFTWARE IMPLEMENTATION

The M2M System Protocol 2 is provided as ANSI-C implementation. The software is composed of the socket interface, protocol stack (NLL/DLL), and platform wrappers. Table 1 lists the header and source files of the M2M System Protocol 2 software.

Table 1. The M2M System Protocol 2 implementation files

<table>
<thead>
<tr>
<th>Module</th>
<th>Header files</th>
<th>Source files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket interface</td>
<td>sp2_control_api.h</td>
<td>sp2_control_api.c</td>
</tr>
<tr>
<td></td>
<td>sp2_socket_api.h</td>
<td>sp2_socket_api.c</td>
</tr>
<tr>
<td></td>
<td>sp2_link_api.h</td>
<td>sp2_link_api.c</td>
</tr>
<tr>
<td></td>
<td>sp2_socket.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sp2_std_socket.h</td>
<td></td>
</tr>
<tr>
<td>Protocol stack</td>
<td>sp2_nll.h</td>
<td>sp2_nll.c</td>
</tr>
<tr>
<td></td>
<td>sp2_dll.h</td>
<td>sp2_dll.c</td>
</tr>
<tr>
<td></td>
<td>sp2_rs.h</td>
<td>sp2_rs.c</td>
</tr>
<tr>
<td></td>
<td>sp2_port_socket.h</td>
<td>sp2_port_socket.c</td>
</tr>
<tr>
<td>Platform wrappers</td>
<td>sp2_system_common.h</td>
<td>sp2_system_common_win32.c</td>
</tr>
<tr>
<td></td>
<td>sp2_dbg_wrap.h</td>
<td>sp2_dbg_wrap_win32.c</td>
</tr>
<tr>
<td></td>
<td>sp2_port.h</td>
<td>sp2_port_os_win32.c</td>
</tr>
<tr>
<td></td>
<td>sp2_wait.h</td>
<td>sp2_port_hw_win32.c</td>
</tr>
<tr>
<td></td>
<td>sp2_types.h</td>
<td></td>
</tr>
</tbody>
</table>

Application software uses the M2M System Protocol 2 functions via the socket interface. An application that uses the M2M System Protocol 2 functionality must include the header file `sp2_socket.h`. In addition, you may add the `sp2_std_socket.h` header to the application source file. The `sp2_std_socket.h` header generalizes the M2M System Protocol 2 namespace to standard names by removing the SP2/sp2-prefix.

The protocol stack provides the core functionality of the M2M System Protocol 2. It is recommended that application software is designed not to use the protocol stack directly.
The platform wrappers contain the platform-specific implementation of the operating system and serial communication services for the M2M System Protocol 2. The platform wrappers must be configured separately for each AM platform.

| Note: The M2M System Protocol 2 file names and external identifiers (except ‘system’ wrappers) are prefixed with ‘sp2’ to avoid name conflicts. |

4.2 BUILDING THE SOFTWARE

This chapter provides a summary of how the M2M System Protocol 2 is built. The instructions are written with the assumption that the reader is already familiar with application software building process in general.

The M2M System Protocol 2 software building process includes the following steps:

   It is up to the application software developer to implement the AM platform wrappers if they are not available. For more information on porting the M2M System Protocol 2, see Chapter 4.3.

2. Compile all the M2M System Protocol 2 source files.
   Compile the selected platform wrappers and all the socket interface/protocol stack source files listed in Table 1. The socket interface and protocol stack source files should compile without modifications.

3. Link the compiled files to application software
   Once the M2M System Protocol source files are compiled (and optionally linked as a library), they are linked with application software to produce the final binary image.

   Tip: Chapter 6.2.2 includes a sample makefile that can be used to get started with building the M2M System Protocol 2 software.

4.3 PORTING THE SOFTWARE

As the M2M System Protocol 2 software is integrated with application software, the AM itself must have enough resources for both the M2M System Protocol 2 and application software. As an example, the resource requirements for the Win32 reference implementation of the M2M System Protocol 2 are listed in Table 2.

<p>| Table 2. M2M System Protocol 2 resource requirements |
| Resource | Requirement |</p>
<table>
<thead>
<tr>
<th>Resource</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>SenderThread, ReceiverThread, MsgQueueThread</td>
</tr>
<tr>
<td>RAM</td>
<td>Static 4 KB (+ thread stack) + dynamic approx. 20 KB</td>
</tr>
<tr>
<td>ROM</td>
<td>Approx. 64 KB</td>
</tr>
</tbody>
</table>

When the M2M System Protocol 2 is started it will create the SenderThread, ReceiverThread, and MsgQueueThread. The SenderThread and ReceiverThread are high priority serial drivers used for packet sending and receiving, respectively. The MsgQueueThread takes care of serial message delivery to/from the M2M System Protocol protocol stack and its priority is application dependent.

The M2M System Protocol 2 memory requirements are approx. 64 KB ROM (without system libraries) and approx. 4 KB RAM. In addition, some memory is needed for dynamic memory and thread stacks. Dynamic memory size ranges from a few kilobytes to a maximum of 30 kilobytes; depending mainly on serial buffer sizes. The memory requirement for each thread stack is a few kilobytes.

**Note:** The *Nokia M2M System Protocol 2 Specification* also defines a minimum implementation for resource-constrained AM devices. The Nokia M2M module supports the minimum implementation.
5. THE M2M SYSTEM PROTOCOL 2 SOCKET INTERFACE

The socket interface is divided into the following APIs: Socket API, M2M System Protocol 2 Control API, and Wireless Link Control API. This chapter describes the APIs and lists the related methods, parameters, and return values. In addition, the chapter provides message sequence diagrams, which illustrate message exchange caused by API calls between the AM and Nokia M2M module.

For more detailed description of the messages, refer to the *Nokia M2M System Protocol 2 Specification*.

**Note:** The M2M System Protocol 2 must be started before making any other M2M System Protocol 2 API calls. For more information on how to start the M2M System Protocol 2, see Chapter 5.2.1.

5.1 SOCKET API

The Socket API provides a standard socket interface. It supports client and server sockets, and UDP/TCP protocols.

5.1.1 Creating a socket (sp2_socket)

The `sp2_socket` function creates a handle to a socket on the AM side of the M2M System Protocol 2. The function call is illustrated in Figure 3.

![](image)

Application software  M2M System Protocol 2  Nokia M2M module software

sp2_socket()  status()  Create a socket

**Figure 3. Creating a socket**

The parameters and return values of the `sp2_socket` function call are described in Table 3 and Table 4.

**Table 3. The sp2_socket function call parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol addressing scheme</td>
<td>IPv4</td>
</tr>
</tbody>
</table>

10/50
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol type</td>
<td>Stream or datagram</td>
</tr>
<tr>
<td>Protocol</td>
<td>TCP or UDP</td>
</tr>
</tbody>
</table>

**Table 4. The sp2_socket function call return values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to the created socket</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Socket allocation failed</td>
</tr>
</tbody>
</table>

An example of creating a socket:

```c
DWORD socket; // a handle to the socket to be created
BYTE addrFormat = SP2_AF_INET; // addressing scheme is IPv4
BYTE type = SP2_SOCK_STREAM; // TCP stream socket
BYTE protocol = SP2_IPPROTO_TCP; // TCP/IP protocol is used
socket = sp2_socket(addrFormat,type,protocol);
if (socket != SP2_SOCKET_ERROR)
```

### 5.1.2 Binding a socket (sp2_bind)

The `sp2_bind` function binds a previously created M2M System Protocol 2 socket to a port. The function call is illustrated in Figure 4.

![Diagram](image)

**Figure 4. Binding a socket**

The parameters and return values of the `sp2_bind` function call are described in Table 5 and Table 6.

**Table 5. The sp2_bind function call parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to a socket to be bound</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>The address of the socket to be bound</td>
</tr>
<tr>
<td>Address length</td>
<td>The length of the socket address in bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Table 6. The sp2_bind function call return values</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return value</strong></td>
</tr>
<tr>
<td>SP2_SOCKET_OK</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
</tr>
</tbody>
</table>

An example of binding a socket:

```c
DWORD socketID; // a handle to the socket
DWORD status; // a return value from the function
struct sp2_sockaddr_in addr; // socket address structure for IPv4

addr.sin_family = SP2_AF_INET; // addressing scheme is IPv4
addr.sin_port = sp2_htons(1234); // port to which the socket is to be bound
addr.sin_addr = sp2_inet_addr("127.0.0.1"); // a socket address

status = sp2_bind(socketID,(sp2_sockaddr *)&addr,sizeof(addr));

if (status == SP2_SOCKET_OK)
```

### 5.1.3 Listening to a socket (sp2_listen)

The **sp2_listen** function creates and binds a socket to the Nokia M2M module, and initiates the listening of the incoming socket connection requests. The function call is illustrated in Figure 5.
Figure 5. Listening to a socket

The parameters and return values of the `sp2_listen` function call are described in Table 7 and Table 8.

Table 7. The `sp2_listen` function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to a socket to be listened</td>
</tr>
<tr>
<td>Queue length</td>
<td>Connection queue length: 0..5</td>
</tr>
</tbody>
</table>

Table 8. The `sp2_listen` function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_SOCKET_OK</td>
<td>Socket was connected successfully</td>
</tr>
</tbody>
</table>
An example of listening to a socket:

```c
DWORD socketID; // a handle to the socket
DWORD status; // a return value from the function
DWORD queueLen = 1; // queue length is 1
status = sp2_listen(socketID, queueLen);
if (status == SP2_SOCKET_OK)
```

### 5.1.4 Accepting a socket (sp2_accept)

The `sp2_accept` function accepts an incoming socket connection. The function will block until an ACCEPT_NOTIFICATION is received from the Nokia M2M module. The function call is illustrated in Figure 6.

**Figure 6. Accepting a socket**

The parameters and return values of the `sp2_accept` function call are described in Table 9 and Table 10.

**Table 9. The sp2_accept function call parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to a socket which is listened to</td>
</tr>
<tr>
<td>Address</td>
<td>The address of the incoming socket</td>
</tr>
<tr>
<td>Address length</td>
<td>The length of the socket address in bytes</td>
</tr>
</tbody>
</table>

**Table 10. The sp2_accept function call return values**
An example of accepting a socket:

```plaintext
DWORD socketID; // a handle to the socket
DWORD acceptedSocketID; // a handle to an incoming socket
struct sp2_sockaddr_in addr; // incoming socket address structure
DWORD addrlen; // incoming socket address length

acceptedSocketID = sp2_accept(socketID,(sp2_sockaddr *)&addr,&addrlen);
if (acceptedSocketID != SP2_SOCKET_ERROR)
```

### 5.1.5 Connecting a socket (sp2_connect)

The `sp2_connect` function connects a previously created M2M System Protocol 2 socket to another socket. The function call is illustrated in Figure 7.

#### Table 11. The `sp2_connect` function call parameters

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to an incoming socket</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Acceptance failure</td>
</tr>
</tbody>
</table>

* In case an error occurs, the function returns with an error status.

**Figure 7. Connecting a socket**

The parameters and return values of the `sp2_connect` function call are described in Table 11 and Table 12.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to a socket to be connected</td>
</tr>
<tr>
<td>Address pointer</td>
<td>The address of a socket to which the existing socket is connected</td>
</tr>
<tr>
<td>Address length</td>
<td>The length of the socket address in bytes</td>
</tr>
</tbody>
</table>

Table 12. The sp2_connect function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_SOCKET_OK</td>
<td>Socket was connected successfully</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Connection failure</td>
</tr>
</tbody>
</table>

An example of connecting a socket:

```c
DWORD socketID; // a handle to the socket
DWORD status; // a return value from the function
struct sp2_sockaddr_in addr; // the address of the socket to be connected
addr.sin_family = SP2_AF_INET; // addressing scheme is IPv4
addr.sin_port = sp2_htons(1234); // port to which socket is to be connected
addr.sin_addr = sp2_inet_addr("127.0.0.1"); // a socket address
status = sp2_connect(socketID,(sp2_sockaddr *)&addr,sizeof(addr));
if (status == SP2_SOCKET_OK)
```

5.1.6 Sending data to a socket (sp2_send / sp2_sendto)

The `sp2_send` and `sp2_sendto` functions write data to a previously connected M2M System Protocol 2 socket. The `sp2_send` function is used with TCP sockets, and the `sp2_sendto` function is used with UDP sockets. Both function calls are illustrated in Figure 8.
The parameters of the \texttt{sp2\_send} function call are described in Table 13.

### Table 13. The \texttt{sp2\_send} function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to a socket to which data is written</td>
</tr>
<tr>
<td>Buffer</td>
<td>Data to be sent</td>
</tr>
<tr>
<td>Data length</td>
<td>The length of the data in bytes</td>
</tr>
<tr>
<td>Flags</td>
<td>Send flags</td>
</tr>
</tbody>
</table>

The parameters of the \texttt{sp2\_sendto} function call are described in Table 14.

### Table 14. The \texttt{sp2\_sendto} function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to the socket to which data is written</td>
</tr>
<tr>
<td>Buffer</td>
<td>Data to be sent</td>
</tr>
<tr>
<td>Data length</td>
<td>The length of the data in bytes</td>
</tr>
<tr>
<td>Flags</td>
<td>Send flags</td>
</tr>
<tr>
<td>Address</td>
<td>The address of the socket to which data is written</td>
</tr>
<tr>
<td>Address length</td>
<td>The length of the socket address in bytes</td>
</tr>
</tbody>
</table>

The return values of both the \texttt{sp2\_send} and \texttt{sp2\_sendto} function calls are described in Table 15.

### Table 15. The \texttt{sp2\_send} and \texttt{sp2\_sendto} function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes written</td>
<td>Number of bytes written successfully</td>
</tr>
<tr>
<td>\texttt{SP2_SOCKET_ERROR}</td>
<td>Failure to write data</td>
</tr>
</tbody>
</table>

An example of sending data to a socket:

```c
DWORD socketID; // the handle to the socket to which data is written
BYTE buff[DATA\_LENGTH]; // buffer to store data to be written
DWORD nbytes = DATA\_LENGTH; // number of bytes to be written
DWORD flags = 0; // no flags
DWORD bytesWritten; // number of bytes written successfully

bytesWritten = sp2\_send(socketID, buff, nbytes, flags);
if (bytesWritten != \texttt{SP2\_SOCKET\_ERROR})
```

---

17/50
5.1.7 Receiving data from a socket (sp2_recv)

The `sp2_recv` function instructs the Nokia M2M module to read data from a socket. The `sp2_recv` function will block and wait until data is available. The function call is illustrated in Figure 9.

![Diagram illustrating the process of receiving data from a socket](image)

*In case an error occurs, the function returns with an error status.

Figure 9. Receiving data from a socket

The parameters and return values of the `sp2_recv` function call are described in Table 16 and Table 17.

Table 16. The `sp2_recv` function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to the socket from which data is read</td>
</tr>
<tr>
<td>Buffer</td>
<td>Buffer for the data to be read</td>
</tr>
<tr>
<td>Data length</td>
<td>The length of the data to be read in bytes</td>
</tr>
<tr>
<td>Flags</td>
<td>Receive flags</td>
</tr>
</tbody>
</table>

Table 17. The `sp2_recv` function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes received</td>
<td>Number of bytes read successfully</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Failure to receive data</td>
</tr>
</tbody>
</table>

An example of receiving data from a socket:
5.1.8 Closing a socket (sp2_close)

The sp2_close function releases a socket handle and instructs the Nokia M2M module to close a socket. The function call is described in Figure 10.

![Diagram of closing a socket](image_url)

**Figure 10. Closing a socket**

The parameters and return values of the sp2_close function call are described in Table 18 and Table 19.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to the socket to be closed</td>
</tr>
</tbody>
</table>

**Table 18. The sp2_close function call parameters**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_SOCKET_OK</td>
<td>Socket was closed successfully</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Failure to close a socket</td>
</tr>
</tbody>
</table>

**Table 19. The sp2_close function call return values**

An example of closing a socket:
5.1.9 Resolving host information (sp2_gethostbyname)

The `sp2_gethostbyname` function resolves host information based on a domain name. The function relies on external Domain Name Service (DNS), so the M2M module must have a wireless link opened to a DNS server. The function call is illustrated in Figure 11.

Figure 11. Resolving host information

The parameters and return values of the `sp2_gethostbyname` function call are described in Table 20 and Table 21.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>A domain name to be resolved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host information</td>
<td>Host information was resolved successfully</td>
</tr>
<tr>
<td>NULL</td>
<td>Failure to resolve a host</td>
</tr>
</tbody>
</table>

An example of resolving host information:

```c
struct hostent *host; // function gethostbyname will reserve memory
const char *hostname = "local"; // a domain name
```
5.1.10 Network utility functions

The M2M System Protocol 2 also provides some utility functions that are typically used when programming sockets. The utility functions are described below.

Convert a 32-bit value from host byte order to network byte order:

```c
DWORD sp2_htonl(DWORD hostlong);
```

Convert a 16-bit value from host byte order to network byte order:

```c
WORD sp2_htons(WORD hostshort);
```

Convert a 32-bit value from network byte order to host byte order:

```c
DWORD sp2_ntohl(DWORD netlong);
```

Convert a 16-bit value from network byte order to host byte order:

```c
WORD sp2_ntohs(WORD netshort);
```

Convert an IPv4 address from a dotted decimal string to 32-bit network byte ordered value:

```c
DWORD sp2_inet_addr(const char *addr);
```

Convert a 32-bit network byte ordered IPv4 address to dotted decimal string:

```c
char *sp2_inet_ntoa(DWORD naddr);
```

5.1.11 Handling socket descriptors (sp2_FD_ZERO, sp2_FD_SET, sp2_FD_CLR, sp2_FD_ISSET)

The socket descriptor functions are used to mask selections for the sp2_select function when using asynchronous sockets.

The sp2_FD_ZERO function resets the socket descriptors of the ‘fdset’ socket descriptor set to the initial SP2_ILLEGAL_SOCKET values:
The `sp2_FD_SET` function inserts an ‘fd’ socket descriptor into the ‘fdset’ socket descriptor set:

```c
void sp2_FD_SET(int fd, void* fdset)
```

The `sp2_FD_CLR` function removes an ‘fd’ socket descriptor from the ‘fdset’ socket descriptor set:

```c
void sp2_FD_CLR(int fd, void* fdset)
```

The `sp2_FD_ISSET` function returns `TRUE` if an event has been set for an ‘fd’ socket descriptor to which the ‘fdset’ socket descriptor set points. Otherwise, the returned value is `FALSE`:

```c
int sp2_FD_ISSET(int fd, void* fdset)
```

### 5.1.12 Reading an asynchronous socket (sp2_select)

The `sp2_select` function indicates which sockets can be read from or written to. In addition, the function indicates the sockets that have pending errors. The function call for accepting a socket is illustrated in Figure 12, and Figure 13 illustrates the function call for receiving data from a socket.

* The `sp2_select` function is looped until it returns an accepted socket count that is greater than zero. That is, the M2M System Protocol 2 indicates that one or more sockets are ready to be accepted.

** The M2M System Protocol 2 may receive an ACCEPT_NOTIF message any time once the server socket is set to listen to incoming connection requests. The M2M System Protocol 2 stores the number of accepted sockets and returns it when the application calls the `sp2_select` function.

Figure 12. Accepting an asynchronous socket
The sp2_select function is looped until it returns a socket count (greater than zero) of sockets that are ready to be read. That is, the M2M System Protocol 2 indicates that one or more sockets are ready to be read.

** The M2M System Protocol 2 must send a RECEIVE message to the M2M module to indicate that it is ready to receive data. The M2M System Protocol 2 sends a RECEIVE message when the application software calls the sp2_select function for the first time. A RECEIVE message is sent once for each socket that is to be read.

*** It is possible that the M2M module has received data before a RECEIVE message is called. The M2M module stores the received data, and returns it to the M2M System Protocol 2 within a RECEIVE_NOTIF message as a response to a RECEIVE message.

Figure 13. Receiving data from an asynchronous socket

The parameters and return values of the sp2_select function are described in Table 22 and Table 23.

Table 22. The sp2_select function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum socket ID</td>
<td>Specifies the range of socket descriptors: from 0 to maximum socket ID – 1.</td>
</tr>
<tr>
<td>Selection read set</td>
<td>Socket descriptor set to mask sockets from which data can be read (sp2_accept, sp2_recv, sp2_recvfrom). Parameter is ignored when NULL.</td>
</tr>
<tr>
<td>Selection write set</td>
<td>Socket descriptor set to mask sockets to which data can be written. Parameter is ignored when NULL. Note that the M2M System Protocol 2 sockets can always be written to.</td>
</tr>
<tr>
<td>Selection error set</td>
<td>Socket descriptor set to mask sockets that have pending errors. Parameter is ignored when NULL.</td>
</tr>
</tbody>
</table>
Table 23. The sp2_select function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ready count</td>
<td>Number of sockets that are ready for asynchronous operations.</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Failure to select.</td>
</tr>
</tbody>
</table>

An example of accepting a socket in asynchronous mode:

```c
int acceptedSocketCount; // how many sockets are ready to be accepted
sp2_fd_set rdset; // initialize using sp2_FD_ZERO and sp2_FD_SET functions
int maxfd; // initialize to the maximum-1 socket ID value
...
while (1) // loop calling the sp2_select function
    acceptedSocketCount = sp2_select(maxfd, &rdset, NULL, NULL);
    if (acceptedSocketCount > 0)
        // use sp2_FD_ISSET to check which sockets have events pending
        // use sp2_accept, sp2_recv, etc. to operate the selected sockets
    // add some code (e.g. yield or sleep) to avoid CPU hogging while looping
```

5.1.13 Controlling socket mode (sp2_ioctl)

The sp2_ioctl function is used to control the mode of a socket. The parameters and return values of the sp2_ioctl function are described in Table 24 and Table 25.

Table 24. The sp2_ioctl function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket ID</td>
<td>A handle to the socket to be controlled.</td>
</tr>
<tr>
<td>Command</td>
<td>The following commands are supported:</td>
</tr>
<tr>
<td></td>
<td>- SP2_FIONBIO: set asynchronous/synchronous mode. A parameter value other than zero indicates asynchronous mode, and a parameter value zero indicates synchronous.</td>
</tr>
<tr>
<td>Value</td>
<td>A value for the command parameter.</td>
</tr>
</tbody>
</table>

Table 25. The sp2_ioctl function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_SOCKET_OK</td>
<td>Socket mode was changed successfully.</td>
</tr>
<tr>
<td>SP2_SOCKET_ERROR</td>
<td>Failure to change a mode.</td>
</tr>
</tbody>
</table>
An example of changing the socket mode to asynchronous:

```
DWORD socketID; // a handle to the socket
DWORD command = SP2_FIONBIO; // asynchronous (non-blocking) mode
DWORD param = 1; // value for command
DWORD status; // a return value from the function

status = sp2_ioctl(socketID, command, param);
if (status == SP2_SOCKET_OK)
```

### 5.1.14 Reading the latest socket status (sp2_socket_status)

The **sp2_socket_status** function is used to discover the actual reason for failed socket operations that only return the general SP2_SOCKET_ERROR. The **sp2_socket_status** function returns the status of the last socket operation. In practice the returned socket status comes either from the response or notification message of the Nokia M2M module.

The **sp2_socket_status** function has no parameters. The return values are described in Table 26.

#### Table 26. The sp2_socket_status function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_SCKT_OK</td>
<td>The defined values are described (without the SP2-prefix) in detail in the Nokia M2M System Protocol 2 Specification.</td>
</tr>
<tr>
<td>SP2_SCKT_ERROR</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ENOTUSER</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EPPFUP</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EBUSY</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EFAULT</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EADDRINUSE</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EADDRNOTAVAIL</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EAFNOSUPPORT</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ECONNREFUSED</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ECONNRESET</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EINVAL</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EISCONN</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EMFILE</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ENETDOWN</td>
<td></td>
</tr>
<tr>
<td>Return value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SP2_SCKT_ENETUNREACH</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ENOBUFS</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ENOPROTOOPT</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EOPNOTSUPP</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EPROTONOSUPPORT</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EPROTOTYPE</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ESOCKNOSUPPORT</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ETIMEDOUT</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ENOTCONN</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ENOTSOCK</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EADDRINVAL</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ELINKID</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ESESSIONID</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_ELIDSIDAA</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EWOULDBLOCK</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_EClosing</td>
<td></td>
</tr>
</tbody>
</table>

An example of reading the status of the last socket operation:

```c
DWORD status; // a return value from the function
status = sp2_socket_status();
if (status != SP2_SCKT_OK) // check for errors
```

## 5.2 M2M SYSTEM PROTOCOL 2 CONTROL API

The M2M System Protocol 2 Control API makes it possible to configure, open and close of a link between the AM and Nokia M2M module.

### 5.2.1 Starting the M2M System Protocol 2 (sp2_start)

The `sp2_start` function is used to establish an M2M System Protocol 2 link between the AM and Nokia M2M module.
The `sp2_start` function must be executed and the `sp2_hLinkReadyEvent` must be received before any other M2M System Protocol 2 API call can be made.

The `sp2_start` function configures the DLL according to the parameters given, and then starts the M2M System Protocol 2 threads. The function call is illustrated in Figure 14.

**Figure 14. Starting the M2M System Protocol 2**

The parameters and return values of the `sp2_start` function call are described in Table 27 and Table 28.

**Table 27. The sp2_start function call parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>M2M System Protocol 2 parameter N1: 50..255</td>
</tr>
<tr>
<td>N3</td>
<td>M2M System Protocol 2 parameter N3: 5..10</td>
</tr>
<tr>
<td>T3</td>
<td>M2M System Protocol 2 parameter T3: 1..15s</td>
</tr>
<tr>
<td>Port</td>
<td>Serial port number used for data link connection</td>
</tr>
<tr>
<td>Baud rate</td>
<td>Baud rate (bit/s) used: 9600, 19200, 38400, 57600, 115200</td>
</tr>
</tbody>
</table>
Note: Typically only the port and baud rate parameters are modified. The recommended values for the N1, N3, and T3 parameters are 255, 10, and 5, respectively. For more information on modifying the N1, N3, and T3 parameter values, see the Nokia M2M System Protocol 2 Specification.

Table 28. The sp2_start function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>M2M System Protocol 2 was started successfully</td>
</tr>
<tr>
<td>FALSE</td>
<td>Failure to start the M2M System Protocol 2</td>
</tr>
</tbody>
</table>

An example of starting a M2M System Protocol 2 link:

```c
BYTE n1 = 255; // maximum data field length: n = 16*(N1+1)
BYTE n3 = 10; // activity timer is 10s
BYTE t3 = 5; // system protocol parameter T3
BYTE port = 1; // use COM port 1
DWORD baudrate = 115200; // serial speed is 115200 bps
DWORD status; // a return value from the function
status = sp2_start(n1,n3,t3,port,baudrate);
if (status == TRUE)
```

5.2.2 Stopping the M2M System Protocol 2 (sp2_stop)

The sp2_stop function is used to stop an M2M System Protocol 2 link between the AM and Nokia M2M module. The function call is described in Figure 15.

Note: After this function is called and the stop event is received, the M2M System Protocol 2 resources are freed, and the protocol must be restarted before issuing any further API requests.
Figure 15. Stopping the M2M System Protocol 2

The `sp2_stop` function call does not have any parameters. The return values are described in Table 29.

**Table 29. The sp2_stop function call return values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>Link was stopped successfully</td>
</tr>
<tr>
<td>FALSE</td>
<td>Failure to stop a link</td>
</tr>
</tbody>
</table>

An example of stopping a M2M System Protocol 2 link:

```c
DWORD status; // a return value from the function
status = sp2_stop();
if (status == TRUE)
```

5.3 WIRELESS LINK CONTROL API

The Wireless Link Control API makes it possible to open and close a pre-configured connection of the Nokia M2M module. Since the configuration cannot be changed via the Wireless Link Control API, the connection configuration must be done either locally using the Configurator software for the Nokia M2M module or remotely over-the-air via the Nokia M2M Gateway.

5.3.1 Opening a wireless link (sp2_link_open)

The `sp2_link_open` function is used to establish a wireless link over a GSM network via the Nokia M2M module. The `sp2_link_open` function call is illustrated in Figure 16.
Figure 16. Opening a wireless link

The parameters of the `sp2_link_open` function call are described in Table 30. The return values from the `sp2_link_open` function call are described in Table 31.

Table 30. The `sp2_link_open` function call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection ID</td>
<td>The Nokia M2M module connection ID to be checked and opened.</td>
</tr>
</tbody>
</table>

Table 31. The `sp2_link_open` function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link ID</td>
<td>Handle to an opened link</td>
</tr>
<tr>
<td>SP2_LINK_ERROR</td>
<td>Failure to open a link</td>
</tr>
</tbody>
</table>

An example of opening a wireless link:

```c
DWORD connectionID = SP2_DEFAULT_CONNECTION; // preconfigured connection ID
DWORD linkID; // a handle to the link that was opened

linkID = sp2_link_open(connectionID);
if (linkID != SP2_LINK_ERROR)
```

* In case an error occurs, the function returns with an error status.
5.3.2 Closing a wireless link (sp2_link_close)

The `sp2_link_close` function is used to close a wireless link over a GSM network via the Nokia M2M module. The function call is depicted in Figure 17.

**Note:** All remote sockets must be closed before closing the wireless link.

<table>
<thead>
<tr>
<th>Application software</th>
<th>M2M System Protocol 2</th>
<th>Nokia M2M module software</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp2_link_close()</td>
<td>LINK_CLOSE()</td>
<td>LINK_NOTIFICATION()</td>
</tr>
<tr>
<td>* err_status / status()</td>
<td>LINK_CLOSE_RESP()</td>
<td>LINK_NOTIFICATION_RESP()</td>
</tr>
<tr>
<td>status()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* In case an error occurs, the function returns with an error status.

Note: If the link is already closed, the `sp2_link_close` call will return immediately, and `LINK_NOTIFICATION` is not used.

**Figure 17. Closing a wireless link**

The parameters and return values of the `sp2_link_close` function call are described in Table 32 and Table 33.

**Table 32. The sp2_link_close function call parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link ID</td>
<td>Link ID to be closed</td>
</tr>
</tbody>
</table>

**Table 33. The sp2_link_close function call return values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_LINK_OK</td>
<td>Link was closed successfully</td>
</tr>
<tr>
<td>SP2_LINK_ERROR</td>
<td>Failure to close a link</td>
</tr>
</tbody>
</table>
5.3.3 Reading a link status (sp2_link_status)

The `sp2_link_status` function returns the status of a wireless link. The `sp2_link_status` function has no parameters. The return values are described in Table 34.

Table 34. The `sp2_link_status` function call return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2_SCKT_NETWORK_OPENED</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_NETWORK_CLOSED</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_NETWORK_DISCONNECTED</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_NETWORK_OPEN_IN_PROGRESS</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_NETWORK_BEARER_NOT_AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_NETWORK_PPP_NEG_FAILED</td>
<td></td>
</tr>
<tr>
<td>SP2_SCKT_NETWORK_RESUMED</td>
<td></td>
</tr>
</tbody>
</table>

The defined values are described (without the SP2-prefix) in detail in the *Nokia M2M System Protocol 2 Specification*.

An example of reading a wireless link status:

```c
DWORD status; // a return value from the function
status = sp2_link_status();
if ((status != SP2_SCKT_NETWORK_OPENED) ||
    (status != SP2_SCKT_NETWORK_OPEN_IN_PROGRESS)) // check for errors
```
6. EXAMPLE OF M2M SYSTEM PROTOCOL 2 SOCKET USAGE

6.1 TYPICAL SOCKET USAGE

Sockets are typically used for client-server communication. This chapter describes the typical use of the M2M System Protocol 2 client and server sockets. Although the sequences are illustrated from an individual socket’s point of view, one M2M System Protocol 2 session and wireless GSM link may, of course, be used for many different sockets.

6.1.1 Using a client socket

Typically application software that uses a M2M System Protocol 2 client socket over wireless connectivity has to:

1. Open a serial link to the Nokia M2M module via the M2M System Protocol 2 Control API.
2. Reset the protocol via the M2M System Protocol 2 Control API.
3. Open a wireless GSM link via the Wireless Link Control API.
4. Communicate with the server application via the Socket API.
5. Close the wireless link via the Wireless Link Control API.
6. Close the serial link to the Nokia M2M module via the M2M System Protocol 2 Control API.

A typical client socket message sequence is illustrated in Figure 18.
6.1.2 Using a server socket

Unlike with client sockets, server sockets do not need to control wireless GSM links. Instead, the Nokia M2M module handles wireless links automatically.

Typically application software that uses a M2M System Protocol 2 server socket has to:

1. Open a serial link to the Nokia M2M module via the M2M System Protocol 2 Control API.

2. Reset the M2M System Protocol 2 via the M2M System Protocol 2 Control API.

3. Wait for an incoming socket request via the Socket API.

Note: In case the wireless link breaks all new and pending socket functions using the link in question will fail. To execute the socket operations that were interrupted, the wireless link has to be re-established and the sockets have to be closed and recreated.
4. Communicate with the client via the Socket API.

5. Close the serial link to the Nokia M2M module via the M2M System Protocol 2 Control API.

A typical server socket message sequence is illustrated in Figure 19.

![Figure 19. A typical server socket message sequence]

**Note:** In case the serial link between the AM and Nokia M2M module is re-established after being broken, the M2M System Protocol 2 will reset the protocol; all pending operations are interrupted, and all sockets are closed.

### 6.2 APPLICATION SOFTWARE EXAMPLE

In this chapter, the Hello World application uses the M2M System Protocol 2 client and server sockets as described in Chapter 6.1. Figure 20 shows an overview of the Hello World application, in which the M2M System Protocol 2 socket interface is used to send a text string "SAY" from the client to the server, and a text string "HELLO WORLD!" back from the server to the client.
6.2.1 Configuring and executing the Hello World application

The Hello World application includes both the client and server implementation. As both the client and server of the Hello World application use the M2M System Protocol 2 sockets, the socket peers are likely to be Nokia M2M modules. Thus, the wireless connection between the two Nokia M2M modules has to be configured so that the client can establish a connection to the server.
When executing the Hello World application, the server must be initialized first; it must be ready to accept incoming socket connections from the client. The Hello World application can be executed, for example, from the command prompt.

Use the following parameters to execute the Hello World server:

```plaintext
> HelloWorld.exe server <com port>
```

Use the following parameters to execute the Hello World client:

```plaintext
> HelloWorld.exe client <com port>
```

When the Hello World application is executed without errors, the client will display the text “Hello World successful!” on the screen. If an error occurs, the client will display an error message with a failure explanation.

### 6.2.2 The Hello World implementation

In the Hello World implementation, the M2M System Protocol 2 serial link is initialized first. When the client receives a link ready indication, it tries to establish a connection to the server that is listening to incoming sockets. After the server has accepted the connection request, the client will write the text string "SAY" to the socket. The server reads the text from its socket and answers with a text string "HELLO WORLD!". Finally, the client receives the server’s greeting through its socket and displays the text on the screen. Listing 1 shows the source code of the Hello World application.

### Listing 1. Hello World – an M2M System Protocol 2 example

```c
// Hello World application: HelloWorld.c

// This is an example of how to use the M2M System Protocol 2 in Win32.
// See the M2M System Protocol 2 Integrator's Manual for more information about this software.
// Usage: HelloWorld.exe client/server <com port>
// Expected output: "Hello World successful!" or an error message
#include <stdio.h>

// The M2M System Protocol 2 Socket interface
#include "sp2_socket.h"

// Size of buffer that holds text messages: "SAY" and "HELLO WORLD!"
#define messageBufferLen 80

static const WORD TcpPortThatServerIsListening = 0x5555;
```
// Client functionality
static BOOL HelloWorldClient(void);
// Server functionality
static BOOL HelloWorldServer(void);

// The function for writing a '\0' char terminated text string to a socket
static BOOL sendMessage(DWORD socket, const char *msg);

// The function for reading a '\0' char terminated string from a socket
static BOOL receiveMessage(DWORD socket, char *messageBuffer, int size);

// Cleanup and exit Hello World (just a convenience function)
static void cleanExit(int statusId, char *statusName);

// This source file has both server and client functionality.
// The first command line argument tells which role is chosen
static BOOL isClient;

// Main entry of the Hello World application
void main(int argc, char * argv[]) {
    // Serial port parameters
    BYTE COMPort;
    LONG COMPortSpeed = 115200;

    if (argc != 3) {
        cleanExit(0,"Usage: helloworld client/server <com port>");
    }

    if (strcmp(argv[1],"client") == 0) {
        isClient = TRUE;
    } else {
        isClient = FALSE; // i.e. isServer
    }
    COMPort = (BYTE)atoi(argv[2]);

    if (isClient) {
        printf("HelloWorld Client at COM port %d\r\n",COMPort);
    } else {
        printf("HelloWorld Server at COM port %d\r\n",COMPort);
    }

    // At first, create the M2M System Protocol 2 hLinkReadyEvent;
    // the hLinkReadyEvent is used by the M2M System Protocol 2 to
    // indicate when the link between the AM and Nokia M2M module is ready.
    if (!system_event_create(&sp2_hLinkReadyEvent) || !sp2_hLinkReadyEvent) {
        cleanExit(10,"ERROR: Cannot create hLinkReadyEvent");
    }

    // The hStopEvent is used by the M2M System Protocol 2 to indicate
    // when the link between the AM and Nokia M2M module is stopped.
    if (!system_event_create(&sp2_hStopEvent) || !sp2_hStopEvent) {
        cleanExit(20,"ERROR: Cannot create hStopEvent");
    }

    // Start the M2M System Protocol 2
    if (!sp2_start(255,10,5,COMPort,COMPortSpeed)) {
        cleanExit(30,"ERROR: Cannot start M2M System Protocol 2");
    }

    // Wait until the M2M System Protocol 2 has started (the timeout is 3
    // seconds)
    if (system_event_wait(sp2_hLinkReadyEvent,3*1000) != SYSTEM_OK) {
        cleanExit(40,"ERROR: M2M System Protocol 2 did not latch to Link Ready");
    }
    // Cleanup and exit
    cleanExit(0,"Exit");
}
cleanExit(40,"ERROR: hLinkReadyEvent not OK");

// When a link is ready the protocol state must be reset.
if (!sp2_reset())
{
cleanExit(50,"ERROR: Cannot reset the M2M System Protocol 2");
}

// Select the role: client or server.
if (isClient)
{
    // Server should be ready before doing client's job
    if (!HelloWorldClient())
    {
        cleanExit(60,"ERROR: HelloWorldClient failed!");
    }
} else
{
    if (!HelloWorldServer())
    {
        cleanExit(60,"ERROR: HelloWorldServer failed!");
    }
}

// Give some time for the system to send messages
Sleep(10*1000);

// Stop the M2M System Protocol 2
if (!sp2_stop())
{
cleanExit(-10,"ERROR: Cannot stop M2M System Protocol 2!");
}

// The hStopEvent indicates that the M2M System Protocol 2 is stopped.
if (system_event_wait(sp2_hStopEvent,30*1000) != SYSTEM_OK)
{
cleanExit(-20,"ERROR: M2M System Protocol 2 was not stopped!");
}

// Close all handles before exiting.
if (!CloseHandle(sp2_hLinkReadyEvent) || !CloseHandle(sp2_hStopEvent))
{
cleanExit(-30,"ERROR: Cannot close all handles!");
}
cleanExit(0,"Hello World successful!");

// The functionality of the Hello World client.
static BOOL HelloWorldClient(void)
{
    DWORD socket;  // A client socket handle
    sp2_sockaddr_in socketAddress;  // The address of the socket
    BOOL success = FALSE;  // Indicates Hello World client's success/failure

    DWORD linkID;  // A wireless link handle
    DWORD connectionID = SP2_DEFAULT_CONNECTION;

    // Please note that wireless link is not needed for local sockets
    if ((linkID = sp2_link_open(connectionID)) == SP2_LINK_ERROR)
    {
        printf("sp2_link_open failed\n");
        return FALSE;
    }

    // Create an M2M System Protocol 2 socket
socket = sp2_socket(SP2_AF_INET, SP2_SOCK_STREAM, SP2_IPPROTO_TCP);
if(socket != SP2_SOCKET_ERROR)
{
    socketAddress.sin_family = SP2_AF_INET;
    socketAddress.sin_port = sp2_htons(TcpPortThatServerIsListening);
    socketAddress.sin_addr = sp2_inet_addr("10.0.0.1");
    if(sp2_connect(socket, (sp2_sockaddr *)&socketAddress, sizeof(socketAddress)) != SP2_SOCKET_ERROR)
    {
        const char *sayString = "SAY";
        printf("Client: send message: %s\n", sayString);
        if (sendMessage(socket, sayString))
        {
            char messageBuffer[messageBufferLen];
            if (receiveMessage(socket, messageBuffer, messageBufferLen))
            {
                printf("Client: got message: %s\n", messageBuffer);
                if (strcmp(messageBuffer,"HELLO WORLD!" ) == 0)
                {
                    success = TRUE;
                }
            }
        }
        else
        {
            printf ("ERROR: Client receiveMessage\n");
        }
    }
    else
    {
        printf ("ERROR: Client sp2_send\n");
    }
}
else
{
    printf ("ERROR: Client sp2_socket\n");
}

// Remember to close the wireless link (if opened earlier)
if (sp2_link_close(linkID) != SP2_LINK_OK)
{
    printf("ERROR: Client sp2_link_close\n");
}
return success;

// The functionality of the Hello World server.
static BOOL HelloWorldServer(void)
{
    DWORD serverSocket; // A server socket handle
    DWORD socket = SP2_SOCKET_ERROR; // A data socket handle from the client
    sp2_sockaddr_in socketAddress; // Socket address structure
    DWORD addrLen; // Socket address length

    // Create an M2M System Protocol 2 socket for the server to start
    // listening to client connections.
    serverSocket = sp2_socket(SP2_AF_INET, SP2_SOCK_STREAM, SP2_IPPROTO_TCP);
    if(serverSocket != SP2_SOCKET_ERROR)
    {
        socketAddress.sin_family = SP2_AF_INET;
socketAddress.sin_port = sp2_htons(TcpPortThatServerIsListening);
socketAddress.sin_addr = 0;

if(sp2_bind(serverSocket, (sp2_sockaddr *)&socketAddress,
sizeof(socketAddress)) != SP2_SOCKET_ERROR)
{
    if(sp2_listen(serverSocket, 0x00) != SP2_SOCKET_ERROR)
    {
        // The server is now ready to receive socket connections
        addrLen = sizeof(socketAddress);
        socket = sp2_accept(serverSocket,
                           (sp2_sockaddr *)&socketAddress, &addrLen);
        if(socket != SP2_SOCKET_ERROR)
        {
            char messageBuffer[messageBufferLen];
            if (receiveMessage(socket, messageBuffer,
                                messageBufferLen))
            {
                printf("Server: got message: \n ", messageBuffer);
                if (strcmp("SAY",messageBuffer) == 0)
                {
                    const char *helloWorldString = "HELLO WORLD!";
                    printf ("Server: send message: \n ", helloWorldString);
                    if (sendMessage(socket, helloWorldString))
                    {
                        // give system time to send message
                        Sleep(3*1000);
                    } else
                    {
                        printf ("ERROR: Server sp2_send
");
                        return FALSE;
                    }
                } else
                {
                    printf ("ERROR: Server receiveMessage
");
                    return FALSE;
                }
            } else
            {
                printf ("ERROR: Server sp2_close
");
                return FALSE;
            }
        } else
        {
            printf ("ERROR: Server sp2_listen
");
            return FALSE;
        }
    } else
    {
        printf ("ERROR: Server sp2_bind
");
        return FALSE;
    }
else
{
    printf ("ERROR: Server sp2_socket
");
    return FALSE;
}
if (sp2_close(serverSocket) == SP2_SOCKET_ERROR)
{
    printf ("ERROR: Server socket sp2_close
");
    return FALSE;
}
return TRUE;
}

// The function for writing a '\0' char terminated text string to a socket
BOOL sendMessage(DWORD socket, const char *msg)
{
    int bytesSent = 0;
    int size = strlen(msg)+1;
    while (bytesSent != size)
    {
        // Send data to the M2M System Protocol 2.
        // Hello World messages are very short and thus probably sent
        // in one request. However, the number of bytes written is checked
        // and sending is looped until all the data has been sent.
        int n = sp2_send(socket, msg+bytesSent, size-bytesSent, 0x00);
        if (n == SP2_SOCKET_ERROR)
        {
            printf("ERROR: sp2_send\n");
            return FALSE;
        }
        bytesSent += n;
        Yield(); // Just to let other threads run
    }
return TRUE;
}

// The function for reading a '\0' char terminated string from a socket
BOOL receiveMessage(DWORD socket, char *messageBuffer, int size)
{
    int bytesReceived = 0;
    BOOL isMessageReceived = FALSE;
    // Read until the '\0' char is received.
    while (!isMessageReceived)
    {
        // Hello World messages are very short and thus probably received
        // in one request. However, the number of bytes read is checked
        // and receiving is looped until all the data has been received.
        int n = sp2_recv(socket, messageBuffer+bytesReceived,
                         size - bytesReceived, 0x00);
        if (n == SP2_SOCKET_ERROR)
        {
            printf("ERROR: sp2_recv\n");
            return isMessageReceived;
        } else
        {
            // Check whether the '\0' char was received.
            int i;
            for (i=bytesReceived; i<bytesReceived+n; i++)
            {
                if (messageBuffer[i] == 0)
                {
                    isMessageReceived = TRUE;
                }
            }
            bytesReceived += n;
        }
        Yield(); // Let other threads run
    }
return isMessageReceived;
}

// Do the cleanup and then exit.
void cleanExit(int statusId, char *statusName)
{
}
To use the M2M System Protocol 2 functions from an application the M2M System Protocol 2 objects must be included within application software. Listing 2 shows a makefile template that lists the Hello World and M2M System Protocol 2 object files.

Listing 2. A sample makefile for the M2M System Protocol 2

```
# Makefile template for the M2M System Protocol 2 (Win32 port)
# List all the objects of the M2M System Protocol 2
# Notice that Win32-specific file names end with "win32"
SP2_OBJS = sp2_control_api.o sp2_socket_api.o sp2_link_api.o sp2_common.o sp2_rs.o sp2_dll.o sp2_nll.o sp2_port_socket.o sp2_port_os_win32.o sp2_port_hw_win32.o sp2_system_common_win32.o sp2_dbg_wrap_win32.o
# Target 'all' contains application software objects instead
```
Tip: The Hello World makefile is a good starting point when integrating the M2M System Protocol 2 with application software.
This chapter provides an example of how to use the Nokia 12 Configurator to configure two Nokia 12 GSM modules as a client and a server. The configuration presented in this chapter has to be carried out in order for the Hello World client to establish a connection to the Hello World server.

The Hello World server configuration
The incoming Challenge Handshake Authentication Protocol (CHAP) for the Point-to-Point Protocol (PPP) must be configured for the server.

1. Select Bearer Selection from the M2M System Mode menu.
2. Define the incoming CHAP username and password (for example, ntn211/ntn211).

An example configuration is shown in Figure 21.

Figure 21. The Hello World server configuration for client authentication
The connection of the server must be configured so that the server is ready to accept incoming requests from the Hello World client. In this example the client will be configured so that it makes a TCP CSD call and that is why the wireless bearer of the server is configured to use TCP CSD.

3. Select Connection 1 from the M2M Bearer Selection dialog.
4. Choose TCP CSD as the wireless bearer.
   An example configuration is shown in Figure 22.

![M2M Bearer Selection](image)

**Figure 22. The Hello World server connection configuration**

You also have to configure the IP address of the server. The IP address that is configured to the server must match with the IP address that the client uses as a parameter for the M2M System Protocol 2 `sp2_connect` function.

5. Select **Cable** from the **IMlet loading** menu.
6. Write 10.0.0.1 as the Nokia 12 IP address.
   An example IP address configuration for the server is shown in Figure 23.
The Hello World client configuration

In the Hello World example, the client is configured so that it makes a direct TCP CSD call to the server. Therefore you need to configure a TCP CSD connection for the client; the client opens the default connection when it connects to the server.

1. Select **Bearer Selection** from the **M2M System Mode** menu.
2. Choose the default connection (for example Connection 1).
An example configuration is shown in Figure 24.
A wireless connection (from the client to the server) has to be configured for the client. The outgoing CHAP configuration of the client must match with the incoming CHAP configuration of the server in order for the server to make a client authentication. In addition, the phone number of the server is defined in the 'Gateway number' field because the client calls the Gateway number to establish a data connection to the server.

3. Select **Connection 1** from the **M2M Bearer Selection** dialog.
4. Choose TCP CSD as the wireless bearer.
5. Define the outgoing CHAP username and password (for example ntn211/ntn211).
6. Define the phone number of the server (for example +123xxxxx).

A wireless connection configuration example for the client is shown in Figure 25.
And finally, you also have to configure the IP address of the client. The IP address that is configured to the client can be any valid host IP address, for example 10.0.0.2.

7. Select **Cable** from the **IMlet loading** menu.
8. Write 10.0.0.2 as the Nokia 12 IP address.

An example IP address configuration for the client is shown in Figure 26.
Figure 26. Hello World client IP address configuration