

USER MANUAL

CAN-ENET Software Support Package

P/N: AX140910

ACRONYMS

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BSD	Berkeley Software Distribution
CAN	Controller Area Network
HTML	HyperText Markup Language
IP	Internet Protocol
LAN	Local Area Network
SSP	Software Support Package

TABLE OF CONTENTS

1	GENERAL INFORMATION.....	4
2	SSP CONTENTS.....	5
2.1	Source Files.....	5
2.2	Examples.....	6
3	DATA TYPES AND CODING STYLE	7
4	USING SSP	8
4.1	Receiving Messages from the Converter.....	8
4.2	Sending Messages to the Converter.....	11
4.3	Discovering the Converter	12
5	DOCUMENTATION	14
6	LICENSE	15
7	VERSION HISTORY.....	16

1 GENERAL INFORMATION

The CAN-ENET Software Support Package (SSP) provides a set of software modules, documentation, and examples for developing application software working with various Axiomatic Ethernet to CAN and Wi-Fi to CAN converters.

The user manual is valid for the SSP with the same two major version numbers as the user manual. For example, this user manual is valid for any SSP version 4.0.x. Updates specific to the user manual are done by adding letters: A, B, ..., Z to the user manual version number.

All SSP software modules are written in a standard C programming language for portability and fully documented. They provide support for Axiomatic proprietary *Communication* and *Discovery* protocols. The *Communication* protocol is mainly used for transmitting CAN messages over Ethernet or any other IP network, and the *Discovery* protocol – for locating the converter on the LAN.

The SSP can be equally used for programming embedded systems with limited resources and for application programming in Windows or Linux.

2 SSP CONTENTS

The SSP is distributed as a zip file with the name: CANenetSSPv<X.X.XY>.zip, where <X.X.X> numbers refer to the SSP main version number and <Y> – to the optional documentation change letter.

To avoid potential issues with displaying the SSP help file, the distribution zip file should be unblocked in Windows if acquired over the internet (after downloading from the Axiomatic website, receiving in e-mail as an attachment, etc.) This can be done by right-clicking the file and pressing the *Unblock* button in *Properties->General->Unblock*.

After extracting the zip archive, the following folder structure will be created:

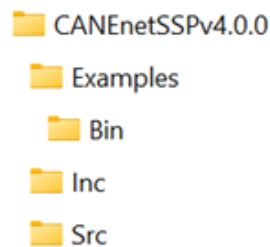


Figure 1. Folder Structure for SSP version 4.0.0

The root directory contains the SSP help file `CANenetSSP.chm` in the Microsoft HTML help format and this user manual `UMAX140910v4.0.pdf` in the Adobe Reader format.

The most significant SSP version number reflects incompatible changes, next – compatible changes, the last one – minor changes not affecting the SSP functionality. The optional letter is added for changes in the user manual and/or help file.

2.1 Source Files

The SSP source files are grouped in `.\Source` and `.\Inc` directories according to their type. They are written in standard C and present the following software modules:

- **PMessage.** Provides support for the protocol independent message structure described in the *Ethernet to CAN Converter Communication Protocol*.
- **CommProtocol.** Supports messages from the *Ethernet to CAN Converter Communication Protocol*.
- **DiscProtocol.** Supports messages from the *Ethernet to CAN Converter Discovery Protocol*.
- **HealthData.** Provides data structures and functions for processing the Ethernet to CAN converter health status information described in the *Ethernet to CAN Converter Communication Protocol*.

All basic data types and common macros are defined in the `CommonTypes.h` file.

2.2 Examples

The SSP also contains the following example programs in the `.\Examples` directory demonstrating different scenarios of communication with the Axiomatic Ethernet to CAN converter:

- `CANReceive.c`. This console application shows how CAN frames can be received from the Axiomatic Ethernet to CAN converter.
- `CANSend.c`. This example demonstrates how CAN frames can be sent to the Axiomatic Ethernet to CAN converter.
- `Discovery.c`. This example application shows how the user can discover an Axiomatic Ethernet to CAN converter on the local area network (LAN).
- `Heartbeat.c`. This application demonstrates how *Heartbeat* messages can be received from the Axiomatic Ethernet to CAN converter. It also shows unpacking of the *Health Data* from *Heartbeat* messages.
- `StatusRequest.c`. This example application shows how the user can request the Axiomatic Ethernet to CAN converter status.

All examples can be built on Microsoft Windows or Linux using `Windows.mk` or `Linux.mk` make files. The make files are also located in the `.\Examples` directory.

Upon building executable files, the make script, if necessary, creates `.\Bin` subdirectory in the `.\Examples` directory where it places all executable and object files. The SSP zip file contains compiled examples for Windows in the `.\Bin` subdirectory.

All SSP examples were tested on Windows 10 and Linux Ubuntu 16.04.

3 DATA TYPES AND CODING STYLE

The SSP uses only `int` and `char` standard data types. The `int` type is used when the exact or maximum data size for the integer parameter is not critical. The `char` type is used to point to an ASCII string or reference a single ASCII character. Other basic types are derived from `<stdint.h>` header and have the exact data size, except the Boolean type `BOOL_t`, which is derived from `int`, see: `CommonTypes.h` file.

All SSP exported basic types are named with capital letters and have the `'_t'` ending. For example: `BOOL_t`, `WORD_t`, etc.

All other exported types are named with capital letters, have the `'_t'` ending and are prefixed with the file abbreviation for the file they are defined in. The `'CP'` is used for the `CommProtocol.h`, `'DP'` - for the `DiscProtocol.h`, `'HD'` - for the `HealthData.h` and `'PM'` - for the `PMessage.h` file.

All macros names use capital letters and are prefixed with the file abbreviation for the file they are defined in, the same way as data types. The `'CT'` abbreviation is used for the `CommonTypes.h` file.

The variable names are prefixed with their type for basic types and pointers. For example: `int` type is prefixed with `'i'`, pointer type - with `'p'`, pointer to integer - with `'pi'`, etc. Structures, unions, enumerators are not prefixed. For zero terminated strings, the `'sz'` prefix is used.

The function names are prefixed with the file abbreviation the same way as data types and macros.

One tab is equal to four spaces.

4 USING SSP

The user should add the SSP files to the application project. The `CommProtocol.c` or `DiscProtocol.c` can be excluded if the appropriate protocol is not used. The `HealthData.c` can also be excluded if there is no need to process the converter health data.

The SSP does not require initialization prior to use. It does not have any global variables. All SSP functions are thread-safe and reentrant.

For sending and receiving converter messages, a support of the Internet protocol (IP) is required. A standard way to provide this support is to use Internet sockets. The socket API is well standardized and is used in all SSP examples and for description of the converter operations.

4.1 *Receiving Messages from the Converter*

The user should first prepare a socket for receiving the converter data.

When the data is received, it should be passed to the `PMParseFromBuffer()` function. The user provides two callback functions: `OnDataParsed()` and `OnDataParsedError()`. The first function is invoked after the protocol message has been successfully parsed and the second one – on the parsing error.

Then, the user should call parsers for individual protocol-specific messages inside the `OnDataParsed()` function, see below:

```
BYTE_t RxData[PM_PROTOCOL_MESSAGE_BUFFER_SIZE];
PM_PROTOCOL_PARSER_t PParser;
int iBytesReceived;

/* Initializing the parser */
memset(&PParser, 0, sizeof(PParser));

/** Receiving data in the RxData buffer.
 * iSocket - the socket descriptor. The socket should be already initialized and ready
 * for receiving.
 */
iBytesReceived = recv(iSocket, RxData, sizeof(RxData), 0);
if(iBytesReceived > 0)
{
    /** Data has been successfully received.
     * Now we are calling the protocol message parser.
     */
    PMParseFromBuffer(RxData, iBytesReceived, &PParser, OnDataParsed, OnDataParsedError,
                      NULL);
}

/* This function is called after the protocol message has been successfully parsed. */
void OnDataParsed(PROTOCOL_MESSAGE_t *pPMessage, void *arg)
{
    DWORD_t dwHealthData, dwCANRxDErrors, dwCANTxDErrors, dwCANBusOffErrors;
    DWORD_t dwMessageNumber, dwTimeInterval;
    CP_CONVERTER_TYPE_t ConverterType;
    DWORD dwCommNodeSupportedFeatures;
    CP_COMMUNICATION_NODE_FILTER_LIST_t CommNodeFilterList;
```

```

/* Parsing Communication Protocol Messages. */

/** Parsing CAN FD Stream. Added in SSP version 3.0.0.
 *
 * The CPParseCANFDStream() parser is provided with a callback function which is
 * called on successful parsing of a CAN FD frame. The CAN FD frame can also contain
 * a Classical CAN frame.
 * The callback functions can be called multiple times if several CAN FD frames are
 * embedded in one protocol message.
 */
if(CPParseCANFDStream(pPMessage, OnCANFDFrameParsed, arg))
{
    /* The CAN FD Stream has been parsed. Add your code here if necessary. */

    return;
}

/** Parsing CAN and Notification Stream. Deprecated in SSP v3.0.0 and used only for
 * compatibility with older software. The new software should use CAN FD Stream
 * with CPParseCANFDStream() parser.
 *
 * The CPParseCANDataAndNotificationStream() parser is provided with two callback
 * functions, which are called on successful parsing of CAN or Notification frames.
 * The callback functions can be called multiple times if several CAN or Notification
 * frames are embedded in one protocol message.
 */
if(CPParseCANDataAndNotificationStream(pPMessage, OnCanFrameParsed,
                                       OnNotificationFrameParsed, arg))
{
    /* The CAN and Notification Stream has been parsed. Add your code here
     if necessary. */

    return;
}

/** Parsing Communication Protocol Status Request Message.
 */
if(CPParseStatusRequest(pPFrame))
{
    /** The Status Request Message has been received.
     *
     * Reply with Status Response Message to let the requesting node know whether
     * the CAN FD stream is supported by your node and convey other communication
     * settings to the requested node. Nodes supporting CAN FD Stream will not start
     * sending CAN frames to your node until they acquire communication settings
     * of your node either through a Status Response or a Heartbeat message.
     *
     * Add your code here.
     */

    return;
}

/** Parsing Communication Protocol Status Response Message.
 */
if(CPParseStatusResponse(pPFrame, &dwHealthData, &dwCANRxDErrors, &dwCANTxDErrors,
                          &dwCANBusOffErrors, &ConverterType,
                          &dwCommNodeSupportedFeatures, &CommNodeFilterList))
{
    /** The Status Response Message has been parsed.
     *
     * dwCommNodeSupportedFeatures and CommNodeFilterList structure contain
     * communication settings of the node.
     * dwCommNodeSupportedFeatures defines whether the node supports CAN FD Stream

```

```

        * (CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM flag is set) and if the node requests
        * only one CAN FD Frame per Ethernet frame
        * (CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM_ONE_FRAME_PER_MESSAGE flag is set)
        * CommNodeFilterList structure contains filter addresses to filter CAN FD
        * Frames sent to the node.
        *
        * Add your code here.
        *
        */

    return;
}

/** Parsing Heartbeat Message.
    */
If(CPParseHeartbeat(pPFrame, &dwMessageNumber, &dwTimeInterval, &dwHealthData,
                    &ConverterType,&dwCommNodeSupportedFeatures,&CommNodeFilterList))
{
    /** The Heartbeat Message has been parsed.
        *
        * dwCommNodeSupportedFeatures and CommNodeFilterList structure contain
        * communication settings of the node.
        * dwCommNodeSupportedFeatures defines whether the node supports CAN FD Stream
        * (CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM flag is set) and if the node requests
        * only one CAN FD Frame per Ethernet frame
        * (CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM_ONE_FRAME_PER_MESSAGE flag is set).
        * CommNodeFilterList structure contains filter addresses to filter CAN FD
        * Frames sent to the node.
        *
        * Add your code here.
        *
        */

    return;
}

/** Unknown protocol message
    *
    */
printf("Error. Unknown protocol message. ProtocolID=%u MessageID=%u\n",
        pPMessage->wProtocolID, pPMessage->wMessageID);
}

/** This function is called after a CAN FD frame has been successfully parsed.
    * The CP_CAN_FD_FRAME_t structure contains either CAN FD or Classical CAN frame.
    */
void OnCANFDFrameParsed(CP_CAN_FD_FRAME_t *pCANFDFrame,CP_CAN_FRAME_ROUTING_DATA_t
                        *pCANFrameRoutingData,DWORD_t dwAbsTimeStamp,void *arg)
{
    /* Add your code here */
}

/** This function is called after a Classical CAN Frame has been successfully parsed.
    Deprecated in SSP v3.0.0
    *
    */
void OnCanFrameParsed(CP_CAN_FRAME_t *pCANFrame,void *arg)
{
    /* Add your code here */
}

/** This function is called after a Notification Frame has been successfully parsed.
    * Deprecated in SSP v3.0.0
    */
void OnNotificationFrameParsed(CP_NOTIFICATION_FRAME_t *pNotificationFrame,void *arg)
{
    /* Add your code here */
}

```

```
}
```

If the user wants to parse the `dwHealthData` value into individual operational statuses of the converter major hardware and software components, the `HDUnpackHealthData()` function should be called:

```
DWORD dwHealthData;
HD_HEALTH_DATA_t HealthData;
CP_CONVERTER_TYPE_t ConverterType;
HD_OPERATIONAL_STATUS_t ConverterHealthStatus;

ConverterHealthStatus = HDUnpackHealthData(dwHealthData, &HealthData, ConverterType);
```

This function also returns the converter aggregated *Health Status*.

4.2 Sending Messages to the Converter

User messages can be sent to the converter by first generating the required protocol message and then copying the message to the transmitting buffer. For example, sending a status request will require the following commands:

```
BYTE_t TxData[PM_PROTOCOL_MESSAGE_BUFFER_SIZE];
PM_PROTOCOL_MESSAGE_t PMessage;
BOOL_t bResult;
int iBytesToSend;
int iBytesSent;

/* Preparing the Status Request message. */
CPGenStatusRequestMessage(&PMessage);
/* Copying the message to the transmit buffer TxData. */
bResult = PMCopyToBuffer(&PMessage, TxData, sizeof(TxData), &iBytesToSend);
assert(bResult);

/* Sending the Status Request message.
   iSocket - the socket descriptor. The socket should be already initialized and
   ready for sending. */
iBytesSent=send(iSocket, TxData, iBytesToSend,0);
```

Sending CAN FD frames is more elaborated. The CAN FD Stream message can contain more than one CAN FD or Classical CAN frame, unless `CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM_ONE_FRAME_PER_MESSAGE` flag is set by the node in the Status Response or Heartbeat message. It is also highly desirable to check whether the node will accept the frame before preparing and sending it.

To send a CAN FD frame, the user should first prepare an empty CAN FD Stream message and then add CAN frames to it.

```
BYTE_t TxData[PM_PROTOCOL_MESSAGE_BUFFER_SIZE];
PM_PROTOCOL_MESSAGE_t PMessage;
CP_CAN_FD_FRAME_t CANFDFrame;
CP_CAN_FRAME_ROUTING_DATA_t CANFrameRoutingData;
DWORD_t dwAbsoluteTimeStamp;
BOOL_t bResult;
int iBytesToSend;
int iBytesSent;

/** Checking whether the CAN FD Stream will be accepted by the node.
 *
```

```

* dwCommNodeSupportedFeatures and CommNodeFilterList values should be already
* acquired from the node in Status Response and Hearbeat messages. If not, the values
* of dwCommNodeSupportedFeatures and CommNodeFilterList should be all 0, and the CAN FD
* Stream will not be sent.
*
*/
if((CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM &
    dwCommNodeSupportedFeatures)==0) return; // CAN FD Stream will not be sent

/* Preparing an empty CAN FD Stream message */
CPPrepareCANFDStream(&PMessage);

/** Adding CAN frames to the CAN FD Stream.
 *
 * CPIsCANFDFrameFit() checks if there is enough room for the CAN FD frame to fit into
 * the CAN FD Stream. If only Classical CAN frames are used, CPIsCANFDFrameFit() can be
 * replaced with CPIsCANClassicalCANFDFrameFit().
 * CANGetFDFrame() gets a CAN FD frame (or a Classical CAN frame in the CAN FD Frame
 * format), CAN frame routing data, and an absolute timestamp (in ms) from a CAN port
 * driver or from another source.
 * CPAddCANFDFrame() adds CAN FD frame to the stream.
 *
 */

while(CPIsCANFDFrameFit(&PMessage))
{
    CANGetFDFrame(&CANFDFrame, &CANFrameRoutingData, &dwAbsoluteTimeStamp);

    if(!CPIsCANAddressPassCommNodeFilter(&CANFrameRoutingData.CANAddr,
        &CommNodeFilterList)) break;

    CPAddCANFDFrame(&PMessage, &CANFDFrame, &CANFrameRoutingData, dwAbsoluteTimeStamp);
    if((CP_SUPPORTED_FEATURE_FLAG_CAN_FD_STREAM_ONE_FRAME_PER_MESSAGE &
        dwCommNodeSupportedFeatures)>0) break;
}

/* Copying the message to the transmit buffer TxData. */
bResult = PMCopyToBuffer(&PMessage, TxData, sizeof(TxData), &iBytesToSend);
assert(bResult);

/** Sending the CAN FD Stream message.
 *
 * iSocket - the socket descriptor. The socket should be already initialized and
 * ready for sending.
 *
 */
iBytesSent=send(iSocket, TxData, iBytesToSend, 0);

```

If the TCP protocol is used, the `TCP_NODELAY` option should be set to the socket to avoid delays in sending protocol messages.

4.3 Discovering the Converter

The converter can be discovered using the *Ethernet to CAN Converter Discovery Protocol*. The user should do the following:

- Open a datagram socket with the `SO_BROADCAST` option.
- Prepare a discovery request and copy it to the transmitting buffer.
- Send the discovery request to the global IP address.
- Wait for the incoming discovery responses from converters located on the same LAN.
- Parse the responses first by `PMParseFromBuffer()` and then by `DPParseResponse()` called from `OnDataParsed()`.

A simplified example code illustrating the concept is presented below:

```
BYTE_t TxData[PM_PROTOCOL_MESSAGE_BUFFER_SIZE];
BYTE_t RxData[PM_PROTOCOL_MESSAGE_BUFFER_SIZE];
PM_PROTOCOL_MESSAGE_t PMessage;
PM_PROTOCOL_PARSER_t PParser;
struct sockaddr_in SocketAddress;

BOOL_t bResult;
int iBytesToSend;
int iBytesSent;
int iBytesReceived;

/* Preparing the Discovery Request message. */
DPGenRequestMessage(&PMessage);
/* Copying the message to the transmit buffer TxData. */
bResult = PMCopyToBuffer(&PMessage, TxData, sizeof(TxData), &iBytesToSend);
assert(bResult);

/* Preparing the global socket address */
memset(&SocketAddress, 0, sizeof(SocketAddress));
SocketAddress.sin_family = AF_INET;
SocketAddress.sin_addr.s_addr = inet_addr("255.255.255.255");
SocketAddress.sin_port = htons(DP_DISCOVERY_PORT);

/* Initializing the parser */
memset(&PParser, 0, sizeof(PParser));

/* Sending the Discovery Request message to the global address.
   iSocket - the socket descriptor. The socket should be already initialized and
   ready for sending to the global address. */
iBytesSent = sendto(iSocket, TxData, iBytesToSend, 0, (struct sockaddr *) &SocketAddress,
                    sizeof(SocketAddress));
if (iBytesSent != SOCKET_ERROR)
{
    /* Now we are waiting for the reply from the converter */

    iBytesReceived = recv(iSocket, RxData, sizeof(RxData), 0);
    if (iBytesReceived > 0)
    {
        /* Reply has been arrived. Parsing it. */
        PMParseFromBuffer(RxData, iBytesReceived, &PParser, OnDataParsed, NULL, NULL);
    }
}

/* This function is called after the protocol message has been successfully parsed. */
void OnDataParsed(PROTOCOL_MESSAGE_t *pPMessage, void *arg)
{
    DP_DISCOVERY_DATA DiscData;

    /* Parsing the Discovery Response Message. */
    if (DPParseResponse(pPMessage, &DiscData))
    {
        /* The Discovery Response Message has been successfully parsed.
           The converter information is in the DiscData structure.
           Add your code here to process this information. */

    }
}
```

5 DOCUMENTATION

The following documents describing the Axiomatic proprietary protocols used in the SSP are available upon request:

- O. Bogush, "Ethernet to CAN Converter Communication Protocol. Document version: 6," Axiomatic Technologies Corporation, April 22, 2025.
- O. Bogush, "Ethernet to CAN Converter Discovery Protocol. Document version: 1A," Axiomatic Technologies Corporation, April 5, 2021.
- O. Bogush, " Ethernet to CAN Converter Health Status. Document version: 4," Axiomatic Technologies Corporation, April 22, 2025.

For requesting the documents, please contact Axiomatic Technologies at:

sales@axiomatic.com

6 LICENSE

The SSP software is distributed with a permissive 3-clause BSD License. The text of the license is included in the software files.

7 VERSION HISTORY

User Manual Version	SSP version	Date	Author	Modifications
4.0A	4.0.x	December 9, 2025	Olek Bogush	<ul style="list-style-type: none"> Updated Axiomatic logo.
4.0	4.0.x	April 23, 2025	Olek Bogush	<ul style="list-style-type: none"> Changed communication node setting parameters in <i>Status Response</i> and <i>Heartbeat</i> messages. Added <i>Generic Converter</i> to the supported converter list. Updated example code and documentation references. Updated CommProtocol.c(h), HealthData.c(h), and examples: CANReceive.c, Heartbeat.c, StatusRequest.c. Updated the front page.
3.0	3.0.x	December 14, 2022	Olek Bogush	<ul style="list-style-type: none"> Added support for CAN FD Stream. Deprecated support for CAN and Notification Stream. Added Communication Node Settings to Status Response and Heartbeat messages. Updated CommProtocol.c, CommProtocol.h, and examples: CANReceive.c, CANSend.c, Heartbeat.c, and StatusRequest.c. Updated Finnish office phone number on the front page.
2.0	2.0.x	April 27, 2021	Olek Bogush	<ul style="list-style-type: none"> Added support for Axiomatic Wi-Fi to CAN converters. Added <i>Converter Type</i> parameter in <i>Heartbeat</i> and <i>Status Response</i> messages. Updated <i>Documentation</i> section. Updated <i>CANReceive.c</i>, <i>Heartbeat.c</i> and <i>StatusRequest.c</i> examples together with <i>Windows.mk</i> and <i>Linux.mk</i> make files.
1.0A	1.0.x	March 2, 2017	Olek Bogush	<ul style="list-style-type: none"> In <i>SSP Contents</i> added request to unblock the distribution .zip file in Windows.
1.0	1.0.x	October 27, 2016	Olek Bogush	<ul style="list-style-type: none"> Initial release.