

Renesas Synergy™ Platform

NetX™, NetX Duo™ BSD Support Module Guide

Introduction

This module guide will enable you to effectively use a module in your own design. Upon completion of this guide, you will be able to add this module to your own design, configure it correctly for the target application, and write code using the included application project code as a reference and an efficient starting point. References to more detailed API descriptions and suggestions of other application projects that illustrate more advanced uses of the module are included in this document and will be valuable resources for creating more complex designs.

The BSD Socket API Compliancy Wrapper (NetX™ BSD) supports a subset of the basic BSD Socket API calls (with some limitations) using NetX™ services.

Note: The NetX Duo BSD is similar to NetX BSD in for BSD application development. When the **NetX BSD** is referenced in this document, the same applies to the **NetX Duo BSD** unless otherwise noted.

This overview covers key elements of the NetX BSD implementation on the Renesas Synergy™ Platform. The primary focus is the addition and configuration of the **NetX BSD Support Module** to a Renesas Synergy™ Platform project. For more details on the operation of this module, consult the *NetX™ BSD 4.3 Socket API Wrapper for NetX User Guide* (or *NetX Duo™ BSD 4.3 Socket API Wrapper for NetX™ Duo User's Guide*) for the Renesas Synergy Platform.

This document is available as part of a X-Ware™ and NetX™ Component Documents for the Renesas Synergy zip file available from the Renesas Synergy Gallery (www.renesas.com/synergy/ssp).

Contents

| | |
|---|----|
| 1. NetX BSD Support Module Features | 2 |
| 2. NetX BSD Support Module APIs Overview | 2 |
| 3. NetX BSD Support Module Operational Overview | 5 |
| 3.1 NetX BSD Support Module Important Operational Notes and Limitations..... | 8 |
| 3.1.1 NetX BSD Support Module Operational Notes | 8 |
| 3.1.2 NetX BSD Support Module Limitations | 9 |
| 4. Including the NetX BSD Support Module in an Application | 10 |
| 5. Configuring the NetX BSD Support Module | 11 |
| 5.1 Configuration Settings for the NetX BSD Support Lower-Level Modules | 12 |
| 5.2 NetX BSD Support Module Clock Configuration | 14 |
| 5.3 NetX BSD Support Module Pin Configuration | 14 |
| 6. Using the NetX BSD Support Module in an Application | 15 |
| 7. The NetX BSD Support Module Application Project..... | 17 |
| 8. Customizing the NetX BSD Support Module for a Target Application | 23 |
| 9. Running the NetX BSD Support Module Application Project..... | 24 |
| 10. NetX BSD Support Module Conclusion..... | 26 |
| 11. NetX BSD Support Module Next Steps..... | 27 |

1. NetX BSD Support Module Features

- The NetX BSD Support module is compliant with BSD 4.3.
- Provides high-level APIs to:
 - Create and delete sockets
 - Set socket options
 - Request TCP connections and listen for connection requests
 - Send and receive packets
 - Raw packet support**
 - PPP over Ethernet support**

** NetX Duo BSD only

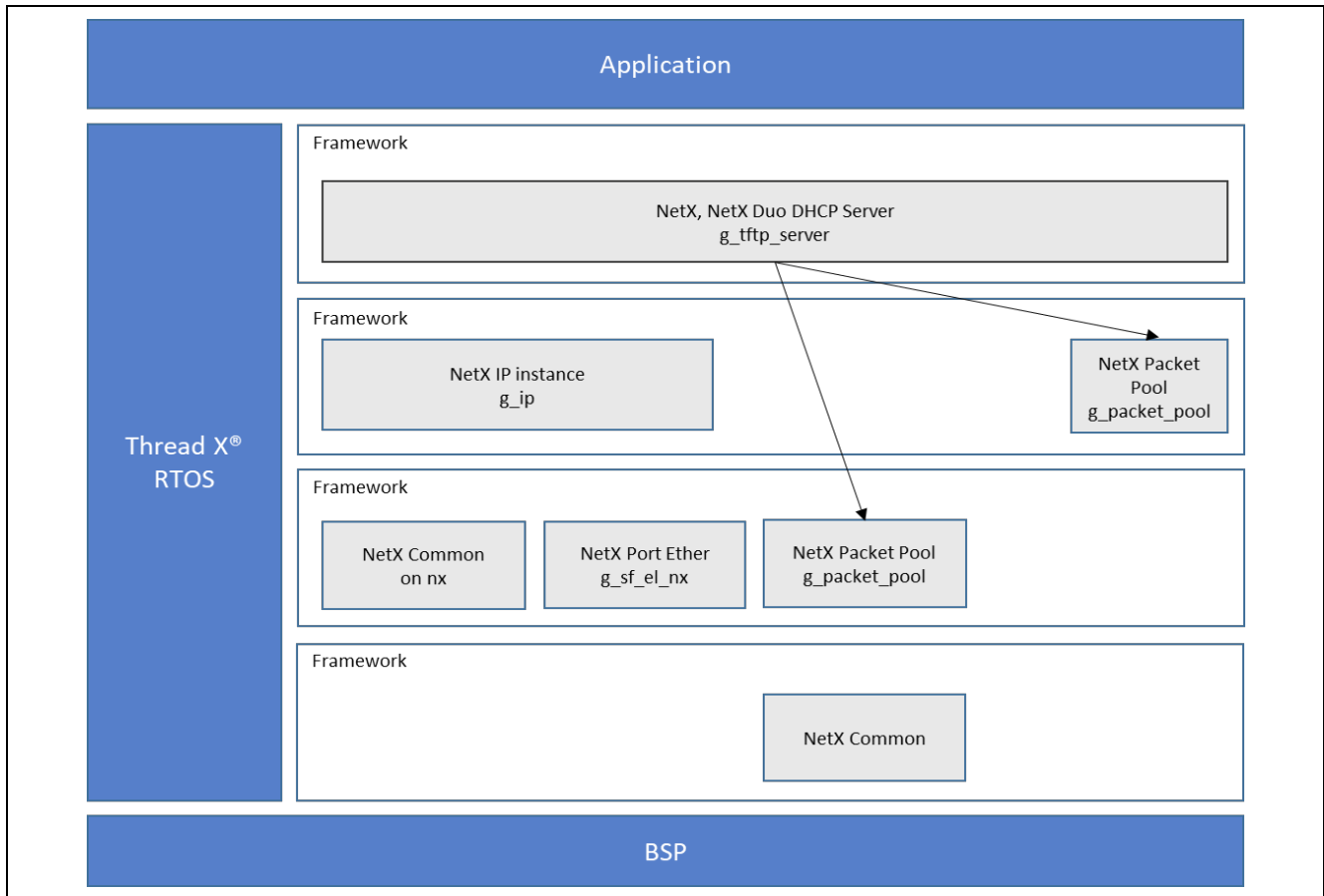


Figure 1. NetX BSD Support Module Block Diagram

2. NetX BSD Support Module APIs Overview

The NetX BSD Support module defines APIs for connecting, binding, listening, sending, and receiving. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table. The services that are unique to NetX and NetX Duo BSD begin with **nx_**, and are listed at the end of this table.

Table 1. NetX BSD Support Module API Summary

| Function Name | Example API Call and Description |
|----------------|--|
| accept | accept(INT sockID, struct sockaddr *ClientAddress, INT *addressLength); TCP server socket waits to make a TCP connection |
| bsd_initialize | bsd_initialize(NX_IP *default_ip, NX_PACKET_POOL *default_pool, CHAR *bsd_thread_stack_area, ULONG bsd_thread_stack_size, UINT |

| Function Name | Example API Call and Description |
|---------------------------|---|
| | <code>bsd_thread_priority);</code> Sets up BSD Support Module to use NetX and BSD services. (Called by the NetX BSD Support Module automatically.) |
| <code>bind</code> | <code>bind(INT sockID, struct sockaddr *localAddress, INT addressLength);</code> Bind TCP or UDP socket to a local port |
| <code>connect</code> | <code>connect(INT sockID, struct sockaddr *remoteAddress, INT addressLength);</code> Connect to a TCP peer; if the remoteAddress indicates raw or UDP socket, then if the address is NULL this dis-associates the peer from this socket. |
| <code>fcntl</code> | <code>fcntl(INT sock_ID, UINT flag_type, UINT f_options);</code> Sets socket options for the specified socket |
| <code>freeaddrinfo</code> | <code>freeaddrinfo(struct addrinfo *res);</code> Releases memory allocated by the getaddrinfo service |
| <code>getnameinfo</code> | <code>getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags);</code> Converts a socket address to a corresponding host and service string. |
| <code>getpeername</code> | <code>getpeername(INT sockID, struct sockaddr *remoteAddress, INT *addressLength);</code> Return remote peer's IP address and port. |
| <code>getsockname</code> | <code>getsockname(INT sockID, struct sockaddr *localAddress, INT *addressLength);</code> Return the socket's primary IP address and source port. In NetX Duo, this would be the address as index 1 in the IP interface table. |
| <code>getsockopt</code> | <code>getsockopt(INT sockID, INT option_level, INT option_name, VOID *option_value, INT *option_length);</code> Return the status of the specified socket option |
| <code>getaddrinfo</code> | <code>getaddrinfo(const CHAR *node, const CHAR *service, const struct addrinfo *hints, struct addrinfo **res);</code> Fills in the addrinfo for the indicated node (host) based on hints in the addrinfo input. |
| <code>ioctl</code> | <code>ioctl(INT sockID, INT command, INT *result);</code> Carry out the command on the specified socket. Only two options supported FIONREAD (extract number of bytes on socket queue) and FIONBIO (enable/disable non blocking as per the result flag). |
| <code>inet_addr</code> | <code>inet_addr(const CHAR *buffer);</code> Convert an IP address from a string buffer to a number. |
| <code>inet_ntoa</code> | <code>inet_ntoa(struct in_addr address_to_convert);</code> Convert an IP address to a string. |
| <code>inet_aton</code> | <code>inet_aton(const CHAR * address_buffer_ptr, struct in_addr *addr);</code> Converts hexadecimal characters into an ASCII IP address representation |
| <code>inet_pton</code> | <code>inet_pton(INT af, const CHAR *src, VOID *dst);</code> Converts an IP address from string to numeric format |
| <code>inet_ntop</code> | <code>inet_ntop(INT af, const VOID *src, CHAR *dst, socklen_t size);</code> |

| Function Name | Example API Call and Description |
|--------------------------------------|--|
| | Converts an IP address from numeric format to string presentation. |
| listen | <code>listen(INT sockID, INT backlog);</code> Bind a TCP server socket to a local port on which to listen for connection requests. |
| recvfrom | <code>recvfrom(INT sockID, CHAR *buffer, INT buffersize, INT flags, struct sockaddr *fromAddr, INT *fromAddrLen);</code> Receive up to the specified number of bytes on the specified socket (either UDP or TCP) |
| recv | <code>recv(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags)</code> Copies up to a specified number of bytes received on the socket into specified location. The given socket can be UDP or TCP, but must be in the connected state. |
| send | <code>send(INT sockID, const CHAR *msg, INT msgLength, INT flags)</code> Send the specified buffer out on the socket; the actual number of bytes sent is returned in msglength. Does not support raw sockets. |
| sendto | <code>sendto(INT sockID, CHAR *msg, INT msgLength, INT flags, struct sockaddr *destAddr, INT destAddrLen);</code> Send the specified buffer out on the socket; the actual number of bytes sent is returned in msglength. The socket must be in a connected state. Supports raw sockets (NetX Duo BSD only) |
| select | <code>select(INT nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);</code> Check all sockets specified in the fd_set inputs to be checked for read request (incoming packets), write request (outgoing packets), and exception request input. |
| soc_close | <code>soc_close(INT sockID);</code> Close the specified socket. |
| socket | <code>socket(INT protocolFamily, INT type, INT protocol);</code> |
| setsockopt | <code>setsockopt(INT sockID, INT option_level, INT option_name, const VOID *option_value, INT option_length);</code> Enable the input socket option on the socket. |
| fcntl | <code>fcntl(INT sock_ID, UINT flag_type, UINT f_options);</code> Sets flag options for the specified socket. |
| nx_bsd_raw_packet_info_extract** | <code>nx_bsd_raw_packet_info_extract(NX_PACKET *packet_ptr, NXD_ADDRESS *nxd_address, UINT *interface_index);</code> Extracts source IP address and interface index of the IP address in the IP interface table |
| nx_bsd_socket_set_inherited_settings | <code>nx_bsd_socket_set_inherited_settings(UINT master_sock_id, UINT secondary_sock_id);</code> Apply the socket options of the specified master socket to the specified child secondary socket; requires <code>NX_BSD_INHERIT_LISTENER_SOCKET_SETTINGS</code> be defined. If it is not, this function has no effect. |

| Function Name | Example API Call and Description |
|--------------------------------------|---|
| <code>nx_bsd_set_service_list</code> | <pre>nx_bsd_set_service_list(struct NX_BSD_SERVICE_LIST *serv_list_ptr, ULONG serv_list_len);</pre> Define the service list used by <code>getaddrinfo</code> with the specified service list. |

** This API is available in NetX Duo BSD only.

The table of NetX BSD API return status codes is omitted. NetX BSD is compliant with the standard BSD and return values defined by the BSD 4.3 specification for the BSD commands.

3. NetX BSD Support Module Operational Overview

To utilize NetX and BSD services, the NetX BSD Support module automatically creates an IP instance, and creates memory space for the internal BSD thread stack. The packet pool can be the IP default packet pool (`g_packet_pool0`) or by clicking on **Add NetX Packet Pool** (or **Add NetX Duo Packet Pool**) -> **New**, a separate packet pool, `g_packet_pool1`, will be used for BSD packet transmissions. For memory space, the defining parameters are the `internal_thread_stack_size` and `internal_thread_priority` properties of the NetX BSD Support stack element.

Before using NetX BSD services, the application creates one or more `sockaddr_in` instances local and peer hosts. A server application will only need to create a `sock_addr` for itself. (These are limited to IPv4 addressing.) For IPv6 addresses (which requires NetX Duo BSD), the application creates one or more `sockaddr_in6` instances. A socket of type TCP or UDP is created using the socket service; the protocol must be `AF_INET` for IPv4 or `AF_INET6` for IPv6.

For raw packets (NetX Duo BSD only), the socket must be of type `AF_PACKET`.

Optionally, the application can set socket options such as non-blocking using the `fcntl` and `ioctl` services.

For TCP and UDP clients, the socket must bind a local-source port using the `bind` service. A value of zero for the port in the `sockaddr_in` the data instance will inform NetX to assign a local port. For the TCP sockets, the client socket connects to a TCP server using the `connect` service.

Both the UDP and TCP sockets can then send and receive data. Because the BSD is a streaming based protocol, data is delivered to and from the application in user-defined buffers. Internally, NetX sends and transmits data using packets from the packet pools transparently to the application. There is no requirement to release received packets or allocated packets that NetX was not able to send.

To close a client socket, the application calls `soc_close`. There is no need to unbind the socket when closing it using `soc_close`.

For TCP servers, the application chooses a local port to listen on by the specified master socket using the `listen` service. The master socket then checks for connection requests using the `select` service. When one is detected, it calls the `accept` service and assigns a secondary socket to handle the connection. In this manner, the BSD can maintain multiple connections simultaneously.

To close a server socket, the application calls `soc_close`. Unlike NetX TCP sockets, there is no need to call the `nx_tcp_server_socket_unlisten` or `nx_tcp_server_socket_unaccept` API on the TCP socket in NetX BSD.

Eliminating an Internal BSD Thread

By default, BSD utilizes an internal thread to perform some of its processing. In systems with tight memory constraints, NetX BSD can be built with `NX_BSD_TIMEOUT_PROCESS_IN_TIMER` defined, which eliminates the internal BSD thread and alternatively uses an internal timer to perform the same processing. This eliminates the memory required for the internal BSD thread control block and stack. The timer processing is significantly increased and the BSD processing may also execute at a higher priority than needed.

To configure BSD sockets to run in the ThreadX-timer context, define `NX_BSD_TIMEOUT_PROCESS_IN_TIMER` in the project. If the BSD layer is configured to execute the BSD tasks in the timer context, the following properties of the BSD stack element are ignored:

- `internal_thread_stack_size`
- `internal_thread_priority`

NetX BSD with DNS Support

If `NX_BSD_ENABLE_DNS` is defined, NetX BSD can send DNS queries to obtain hostname or host IP information. This feature requires a NetX DNS Client instance to be previously created. The BSD link to this DNS instance is via an extern `NX_DNS *_nx_dns_instance_ptr`. When the BSD application calls **getnameinfo** with an address or **getaddrinfo** with a hostname, NetX BSD will call various NetX DNS Client services to obtain the host name or IP address, respectively. Refer to the *NetX DNS Client User Guide* for the Renesas Synergy Platform, available as described at the Reference section at the end of this document, for more details on setting up a DNS Client in an application.

Raw Socket Support (NetX Duo BSD only)

To use raw sockets in the NetX Duo BSD, the NetX Duo library must be compiled with `NX_ENABLE_IP_RAW_PACKET_FILTER` defined in the project. By default, it is not defined. To define it, right click on the project in the **Synergy Configuration** pane -> **Properties** -> **Cross ARM C Compiler** -> **Preprocessor** and add this to the list of defined symbols by clicking on the (+) icon.

The application must then enable raw socket processing for a previously created IP instance by calling the `nx_ip_raw_packet_enable` service before using NetX Duo BSD services. To create a raw socket in NetX Duo BSD, the application uses the socket service and specifies the protocol family, socket type, and protocol:

```
sock_l = socket(INT protocolFamily, INT socket_type, INT protocol)
```

The NetX BSD supports these values for protocolFamily : `AF_INET` for IPv4 sockets, `AF_INET6` for IPv6 sockets, and `AF_PACKET` for raw sockets. The socket_type must be set to `SOCK_RAW`. The protocol is application specific.

To send and receive raw packets as well as close a raw socket, the application typically uses the same BSD services as the UDP like `sendto`, `recvfrom`, `select`, and `soc_close`. Raw sockets do not support either accept or listen BSD services.

- By default, received IPv4 raw data includes the IPv4 header. Conversely, received IPv6 raw data does not include the IPv6 header.
- By default, when sending either raw IPv6 or IPv4 packets, the BSD wrapper layer adds the IPv6 or IPv4 header before sending the data.

The NetX Duo BSD supports additional raw socket options, including `IP_RAW_RX_NO_HEADER`, `IP_HDRINCL` and `IP_RAW_IPV6_HDRINCL`. If `IP_RAW_RX_NO_HEADER` is set, the IPv4 header is removed so that the received data does not contain the IPv4 header, and the reported message length does not include the IPv4 header. For IPv6 sockets, by default the raw socket receive does not include the IPv6 header, equivalent to having the `IP_RAW_RX_NO_HEADER` option set. The application may use the `setsockopt` service to clear the `IP_RAW_RX_NO_HEADER` option. Once the `IP_RAW_RX_NO_HEADER` option is cleared, the received IPv6 raw data would include the IPv6 header, and the reported message length includes the IPv6 header. This option has no effect on IPv4 or IPv6 transmitted data.

If `IP_HDRINCL` is set, the application includes the IPv4 header when sending data. This option has no effect on IPv6 transmission and is not defined by default. If the `IP_RAW_IPV6_HDRINCL` is set, the application includes the IPv6 header when sending data. This option has no effect on IPv4 transmission and is not defined by default.

`IP_HDRINCL` and `IP_RAW_IPV6_HDRINCL` have no effect on IPv4 or IPv6 reception.

Note: The BSD 4.3 Socket specification specifies that the kernel must copy the raw packet to each socket receive buffer. However, in NetX Duo BSD, if multiple sockets are created sharing the same protocol, the behavior is undefined.

Raw Packet Support for PPPoE (NetX Duo BSD only)

To enable the raw packet support for PPPoE, `NX_BSD_RAW_PPPOE_SUPPORT` must be defined in the project. To define it, right click on the project in the **Synergy Configuration** pane -> **Properties** -> **Cross**

ARM C Compiler -> Preprocessor and add this to the list of defined symbols by clicking on the (+) icon. This does not require that the NetX Duo library be rebuilt.

The following command creates a BSD socket to handle PPPoE raw packets:

```
sockfd = socket(AF_PACKET, SOCK_RAW, protocol);
```

The current BSD implementation only supports two protocol types in `AF_PACKET` family:

- `ETHERTYPE_PPPOE_DISC`: PPPoE Discovery packets. In the MAC data frame, the Discovery packets have the Ethernet frame type 0x8863.
- `ETHERTYPE_PPPOE_SESS`: PPP Session packets. In the MAC data frame, the Session packets have the Ethernet frame type 0x8864.

The data type `sockaddr_ll` (the `ll` stands for link layer) is used to specify parameters when sending or receiving PPPoE frames.

`struct sockaddr_ll` is declared as:

```
struct sockaddr_ll
{
USHORT      sll_family; /* Must be AF_PACKET */
USHORT      sll_protocol; /* LL frame type */
INT         sll_ifindex; /* Interface Index. */
USHORT      sll_hatype; /* Header type */
UCHAR       sll_pkttype; /* Packet type */
UCHAR       sll_halen; /* Length of address */
UCHAR       sll_addr[8]; /* Physical layer address */
};
```

Note that not every field in the structure is used by `sendto()` or `recvfrom()`. See the following description on how to set up the `sockaddr_ll` for sending and receiving PPPoE packets.

A socket created in the `AF_PACKET` family can be used to send either PPPoE discovery packets or PPP session packets, regardless of the protocol specified. When transmitting a PPPoE packet, application must prepare the buffer that includes properly formatted PPPoE frame, including the MAC headers (destination MAC address, source MAC address, and frame type.) The size of the buffer includes the 14-byte Ethernet header.

In the `sockaddr_ll` struct, the `sll_ifindex` is used to indicate the physical interface to be used for sending this packet. The rest of the fields in the structure are not used. Values set to the unused fields are ignored by the BSD internal process.

The following code block illustrates how to transmit a PPPoE packet:

```
struct sockaddr_ll peer_addr;
/* Transmit the PPPoE frame using the primary network interface. */
peer_addr.sll_ifindex = 0;
n = sendto(sockfd, frame, frame_size, 0, (struct
sockaddr*)&peer_addr, sizeof(peer_addr));
```

The return value indicates the number of bytes transmitted. Since PPPoE packets are message-based, on a successful transmission, the number of bytes sent matches the size of the packet, including the 14-byte Ethernet header.

PPPoE packets can be received using `recvfrom()`. The receive buffer must be big enough to accommodate a message of ethernet MTU size. The received PPPoE packet includes a 14-byte ethernet header. On receiving PPPoE packets, PPPoE discovery packets can only be received by socket created with protocol `ETHERTYPE_PPPOE_DISC`. Similarly, PPP session packets can only be received by socket created with protocol `ETHERTYPE_PPPOE_SESS`. If multiple sockets are created for the same protocol type, arriving PPPoE packets are forwarded to the socket created first. If the first socket created for the protocol is closed, the next socket in the order of creation is used for receiving these packets.

After a PPPoE packet is received, the following fields in the `sockaddr_ll` structure are valid:

- `sll_family`: Set by the BSD internal to be `AF_PACKET`
- `sll_ifindex`: Indicates the interface from which the packet is received
- `sll_protocol`: Set to the type of packet received: `ETHERTYPE_PPPOE_DISC` or `ETHERTYPE_PPPOE_SESS`

3.1 NetX BSD Support Module Important Operational Notes and Limitations

3.1.1 NetX BSD Support Module Operational Notes

NetX BSD Build Requirements

- Add the NetX Source element to the configuration and set the **Extended Notify Support** property to **enabled**.
- Define the following property in the ThreadX Source element:


```
— TX_THREAD_EXTENSION_2    int bsd_errno;
```

 By default, all the `EXTENSION` macros are undefined; `bsd_errno` can also be defined as `TX_THREAD_EXTENSION_3`, `TX_THREAD_EXTENSION_1` or `TX_THREAD_EXTENSION_0`.
- On some systems, there may be a conflict with definition of types in the native BSD. If this happens, include `_POSIX_SOURCE` among the project preprocessor definitions. This is done by right clicking at the top level of the project, **Properties -> C/C++ Build-> Settings -> Cross ARM C Compiler** (if using that project platform) -> **Preprocessor**.

NetX BSD Socket Options

NetX BSD socket options can be enabled (or disabled) at run time on a per socket basis using the `setsockopt` service, which takes as one of its inputs `option_level`. There are two different settings for `option_level`. The first type is `SOL_SOCKET` for socket level options. The list of supported options are:

`SO_BROADCAST`

If set, this enables sending and receiving broadcast packets from NetX sockets. This is the default behavior for NetX Duo. All sockets have this capability.

`SO_ERROR`

Used to obtain socket status on the previous socket operation of the specified socket, using the `getsockopt` service. All sockets have this capability.

`SO_KEEPA_LIVE`

If set, this enables the TCP Keep Alive feature. This requires the NetX Duo library to be built with `NX_TCP_ENABLE_KEEPA_LIVE` defined (setting the **TCP Keepalive** property to enabled in the NetX and NetX Duo Common stack element). By default, this feature is disabled.

`SO_RCVTIMEO`

This sets the wait option in seconds for receiving packets on NetX Duo BSD sockets. The default value is the `NX_WAIT_FOREVER` (0xFFFFFFFF) or, if non-blocking is enabled, `NX_NO_WAIT` (0x0).

`SO_RCVBUF`

This sets the window size of the TCP socket. The default value, `NX_BSD_TCP_WINDOW`, is set to 64 k for BSD TCP sockets. To set the size over 65535 requires the NetX Duo library to be built with the `NX_TCP_ENABLE_WINDOW_SCALING` be defined (setting the `TCP Keepalive` property to enabled in the NetX and NetX Duo Common stack element).

`SO_REUSEADDR`

If set, this enables multiple sockets to be mapped to one port. The typical usage is for the TCP Server socket. This is the default behavior of NetX sockets.

Note: The `SO_ERROR` option requires that the `bsd_errno` is defined. To define `bsd_errno`, add a **NetX Source Stack** below the **NetX Common** on the `nx` element. Or if using **NetX Duo**, add a **NetX Duo Source** element below the **NetX Duo Common** on the `nxd` element. Then add a **ThreadX Source Stack** element in the **NetX Duo Source**. Scroll down the list of **Properties** and choose one of the `TX_THREAD_EXTENSION` macros (0-2). Set the value as follows:

```
int bsd_errno;
```


This is explained in NetX BSD Build Requirements section above.

The other type is `IP_PROTO` for options that are implemented at the IP layer and affect all sockets. The list of run time IP level options is shown below:

`IP_MULTICAST_TTL`

This sets the time to live for UDP sockets. The default value is `NX_IP_TIME_TO_LIVE (0x80)` when the socket is created. This value can be overridden by calling `setsockopt` with this socket option before calling the socket service.

`IP_ADD_MEMBERSHIP`

If set, this option enables the BSD socket (applies only to UDP sockets) to join the specified IGMP group.

`IP_DROP_MEMBERSHIP`

If set, this option enables the BSD socket (applies only to UDP sockets) to leave the specified IGMP group.

The following options are **only supported** in **NetX Duo BSD**:

`IP_HDRINCL`

If this option is set, the calling application must append the IP header and optionally application headers to data being transmitted on raw IPv4 sockets created in BSD. To use this option, raw socket processing must be enabled on the IP task. See the previous section **Raw Socket Support** for specific details.

`IP_RAW_IPV6_HDRINCL`

If this option is set, the calling application must append an IPv6 header and optionally application headers to data being transmitted on raw IPv6 sockets created by BSD. To use this option, raw socket processing must be enabled on the IP task. See the previous section **Raw Socket Support** for specific details.

`IP_RAW_RX_NO_HEADER`

If cleared, the IPv6 header is included with the received data for raw IPv6 sockets created in BSD. IPv6 headers are removed by default in BSD raw IPv6 sockets, and the packet length does not include the IPv6 header. If set, the IPv4 header is removed from received data on BSD raw sockets of type IPv4. IPv4 headers are included by default in BSD raw IPv4 sockets and packet length includes the IPv4 header. This option has no effect on either IPv4 or IPv6 transmission data. See the previous section **Raw Socket Support** for specific details on enabling raw packet support.

To retrieve an option setting, call `getsockopt` for the option name with `option_level` again set to `SOL_SOCKET` for socket level options or `IP_PROTO` for IP level options.

More details on run time socket level options are available in the *NetX™ BSD 4.3 Socket API Wrapper for NetX User's Guide* for the Renesas Synergy™ Platform and *NetX Duo™ BSD 4.3 Socket API Wrapper for NetX Duo User's Guide* for the Renesas Synergy™ Platform documents available as described previously in the Introduction section.

3.1.2 NetX BSD Support Module Limitations

- Only `MSG_DONTWAIT` and `MSG_PEEK` flags are supported for `send`, `recv`, `sendto` and `recvfrom` calls.
- NetX BSD socket level options are limited to:
 - `SO_BROADCAST`
 - `SO_ERROR`
 - `SO_KEEPALIVE`
 - `SO_RCVTIMEO`
 - `SO_RCVBUF`
 - `SO_REUSEADDR`
- NetX BSD IP level options are limited to:
 - `IP_MULTICAST_TTL`
 - `IP_RAW_IPV6_HDRINCL` (NetX Duo BSD only)

- IP_ADD_MEMBERSHIP
- IP_DROP_MEMBERSHIP
- IP_HDRINCL (NetX Duo BSD only, raw sockets must be enabled)
- IP_RAW_RX_NO_HEADER (NetX Duo BSD only, raw sockets must be enabled)
- Refer to the most recent *SSP Release Notes* for any additional operational limitations for this module.

4. Including the NetX BSD Support Module in an Application

This section describes how to include the NetX BSD Support Module in an application using the SSP configurator.

Note: This section assumes you are familiar with creating a project, adding threads, adding a stack to a thread, and configuring a block within the stack. If you are unfamiliar with any of these items, refer to the first few sections of the *SSP User's Manual* to learn how to manage each of these important steps in creating SSP-based applications.

To add the NetX BSD Support Module to an application, add it to a thread using the stacks selection sequence given in the following table.

Table 2. NetX and NetX Duo BSD Support Module Selection Sequence

| Resource | ISDE Tab | Stacks Selection Sequence |
|----------------------|----------|--|
| NetX BSD Support | Threads | New Stack> X-Ware> NetX> Protocols> NetX BSD Support |
| NetX Duo BSD Support | Threads | New Stack> X-Ware> NetX Duo> Protocols> NetX Duo BSD Support |

When the NetX BSD Support module added to the thread stack as shown in the following figure, the configurator automatically adds any needed lower-level drivers. Any drivers that need additional configuration information will be box text highlighted in red. Modules with a gray band are individual modules that stand alone. Modules with a blue band are shared or common and need only be added once and can be used by multiple stacks. Modules with a pink band can require the selection of lower-level drivers; these are either optional or recommended. (This is indicated in the block with the inclusion of this text.) If the addition of lower-level drivers is required, the module description will include **Add** in the text. Clicking on any pink banded modules will bring up the **New** icon and then display the possible choices.

The NetX BSD needs a packet pool for sending packets. You can use the same packet pool created for the NetX IP instance `g_packet_pool0` as shown in the following figure. Another option is to create a new packet pool for NetX BSD and fill in the properties box to specify the size of the payload and the number of packets. A separate packet pool might be a more optimal use of memory if the expected file transfer packets will require a different payload from the IP default packet pool. The latter is typically also shared with the driver to receive packets.

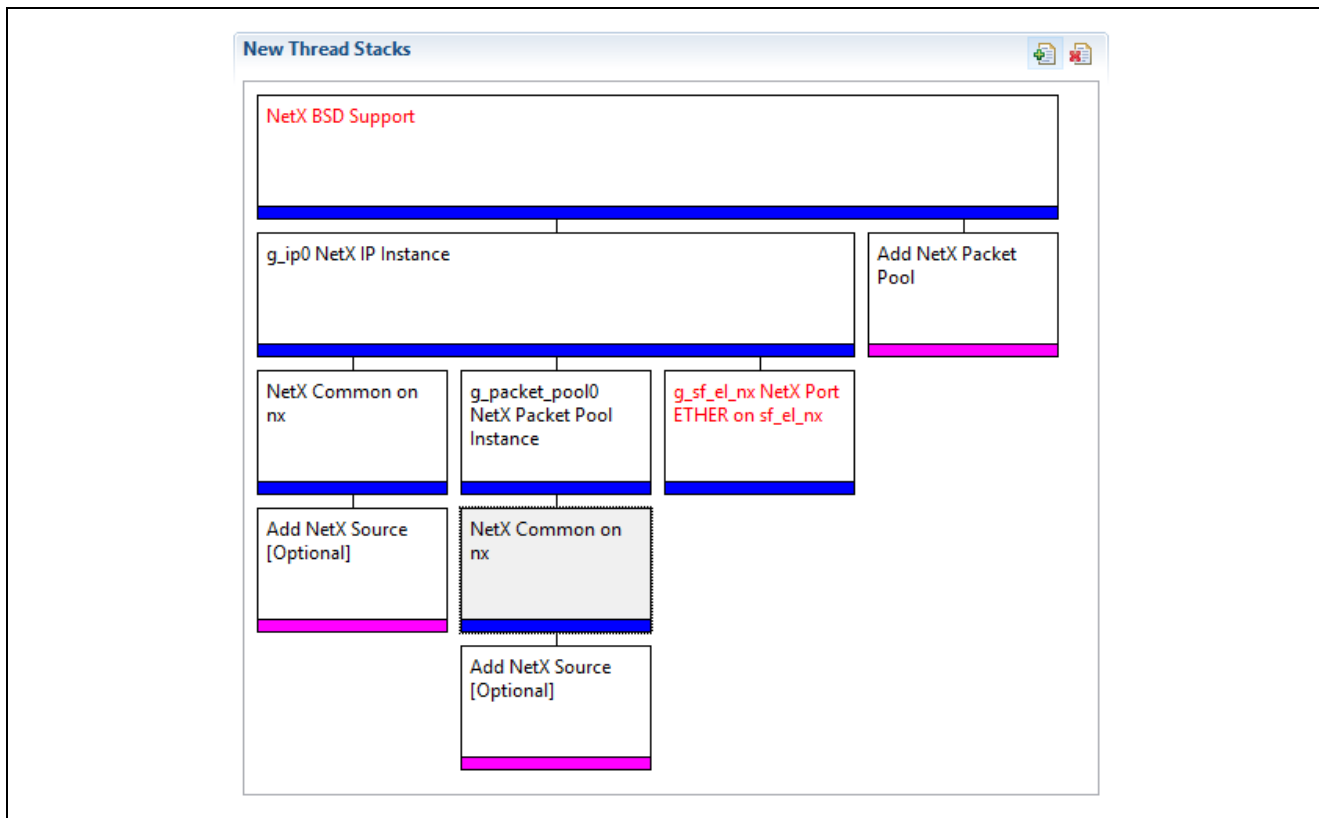


Figure 2. NetX BSD Support Module Stack

5. Configuring the NetX BSD Support Module

The NetX BSD Support module must be configured by the user for the desired operation. The SSP configuration window will automatically identify (by highlighting the block in red) any required configuration selections, such as interrupts or operating modes, which must be configured for lower-level modules for successful operation. Only those properties that can be changed without causing conflicts are available for modification. Other properties are **locked** and are not available for changes, and are identified with a lock icon for the **locked** property in the **Properties** window in the ISDE. This approach simplifies the configuration process and makes it much less error-prone than previous **manual** approaches to configuration. The available configuration settings and defaults for all the user-accessible properties are given in the properties tab within the SSP Configurator and are shown in the following tables for easy reference.

One of the properties most often identified as requiring a change is the interrupt priority; this configuration setting is available within the **Properties** window of the associated module. Simply select the indicated module and then view the Properties window; the interrupt settings are often toward the bottom of the properties list, so scroll down until they become available. Note that the interrupt priorities listed in the Properties window in the ISDE will include an indication as to the validity of the setting based on the targeted MCU (CM4 or CM0+). This level of detail is not included in the following configuration properties tables, but is easily visible within the ISDE when configuring interrupt-priority levels.

Note: You may want to open your ISDE, create the module, and explore the property settings in parallel with looking over the following configuration table settings. This will help orient you and can be a useful **hands-on** approach to learning the ins and outs of developing with SSP.

Table 3. Suggested Configuration Settings for the NetX BSD Support Module

| ISDE Property | Value | Description |
|-----------------------------------|-------|--------------------------------------|
| Internal thread stack size(bytes) | 2048 | Internal thread stack size selection |
| Internal thread priority | 3 | Internal thread priority selection |

Note: The example settings and defaults are for a project using the S7G2 Synergy MCU Group. Other MCUs may have different default values and available configuration settings.

In some cases, settings other than the defaults for lower-level modules can be desirable. For example, it might be useful to select different MAC or IP Addresses. The configurable properties for the lower-level stack modules are given in the following sections for completeness and as a reference.

Note: Most of the property settings for lower-level modules are intuitive and usually can be determined by inspection of the associated Properties window from the SSP configurator.

5.1 Configuration Settings for the NetX BSD Support Lower-Level Modules

Only a small number of settings must be modified from the default for lower-level modules as indicated via the red text in the thread stack block. Notice that some of the configuration properties must be set to a certain value for proper framework operation and will be locked to prevent modification. The following table identifies all the settings within the Properties section for the module. **Entries with yellow highlight must be changed to match your system.** **Entries with a green highlight display a recommended value different from the default value.**

Table 4. Configuration Settings for the NetX IP Instance

| ISDE Property | Value | Description |
|--|---|--|
| Name | g_ip0 | Module name |
| IPv4 Address (use commas for separation) | 192,168,0,2 | IPv4 Address selection |
| Subnet Mask (use commas for separation) | 255,255,255,0 | Subnet Mask selection |
| **IPv6 Global Address (use commas for separation) only if using BSD over IPv6 | 0x2001, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1 | IPv6 global address selection |
| **IPv6 Link Local Address (use commas for separation, all zeros means use MAC address) only if using BSD over IPv6 | 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 | IPv6 link local address selection |
| IP Helper Thread Stack Size (bytes) | 2048 (default is 1024) | IP Helper Thread Stack Size (bytes) selection. Recommended this priority be the highest as any thread using NetX services. |
| IP Helper Thread Priority | 1 (default is 3) | IP Helper Thread Priority selection |
| ARP | Enable | ARP selection |
| ARP Cache Size in Bytes | 520 | ARP Cache Size in Bytes selection |
| Reverse ARP | Enable, Disable Default: Disable | Reverse ARP selection |
| TCP | Enable, Disable Default: Enable | TCP selection |
| UDP | Enable, Disable Default: Enable | UDP selection |
| ICMP | Enable, Disable Default: Enable | ICMP selection |
| IGMP | Enable, Disable Default: Enable | IGMP selection |
| IP fragmentation | Enable, Disable Default: Disable | IP fragmentation selection |

Note: The example settings and defaults are for a project using the S7G2 Synergy MCU Group. Other MCUs may have different default values and available configuration settings.

****Indicates properties only included with NetX Duo**

Table 5. Configuration Settings for the NetX Packet Pool Instance (when added as new)

| ISDE Property | Value | Description |
|---------------------------|--------------------------|--|
| Name | g_packet_pool1 | Module name |
| Packet Size in Bytes | 1568 (default 640 bytes) | Packet size selection. This eliminates the need for packet chaining of packets exceeding the packet payload. |
| Number of Packets in Pool | 16 | Number of packets in pool selection. For very busy networks, this may need to be increased. |

Note: The example settings and defaults are for a project using the S7G2 Synergy MCU Group. Other MCUs may have different default values and available configuration settings.

Table 6. Configuration Settings for the NetX Packet Pool Instance

| ISDE Property | Value | Description |
|---------------------------|--------------------------|--|
| Name | g_packet_pool0 | Module name |
| Packet Size in Bytes | 1568 (default 640 bytes) | Packet size selection. This eliminates the need for packet chaining of packets exceeding the packet payload. |
| Number of Packets in Pool | 16 | Number of packets in pool selection. For very busy networks, this may need to be increased. |

Note: The example settings and defaults are for a project using the S7G2 Synergy MCU Group. Other MCUs may have different default values and available configuration settings.

Entries in yellow indicate settings that must be changed from the default value for the SK-S7G2 processor

Table 7. Configuration Settings for the NetX Common Instance

| ISDE Property | Value | Description |
|--------------------------|-------|-------------|
| No configurable settings | | |

Note: The example settings and defaults are for a project using the S7G2 Synergy MCU Group. Other MCUs may have different default values and available configuration settings.

Table 8. Configuration Settings for the NetX Port ETHER

| ISDE Property | Value | Description |
|---------------------------------------|--|---|
| Parameter Checking | BSP, Enabled, Disabled Default: BSP | Enable or disable the parameter checking |
| Channel 0 Phy Reset Pin | IOPORT_PORT_09_PIN_03 | Channel 0 Phy reset pin selection |
| Channel 0 MAC Address High Bits | 0x00002E09 | Channel 0 MAC address high bits selection |
| Channel 0 MAC Address Low Bits | 0x0A0076C7 | Channel 0 MAC address low bits selection |
| Channel 1 Phy Reset Pin | IOPORT_PORT_08_PIN_06 | Channel 1 Phy reset pin selection |
| Channel 1 MAC Address High Bits | 0x00002E09 | Channel 1 MAC address high bits selection |
| Channel 1 MAC Address Low Bits | 0x0A0076C8 | Channel 1 MAC address low bits selection |
| Number of Receive Buffer Descriptors | 8 | Number of receive buffer descriptors selection |
| Number of Transmit Buffer Descriptors | 32 | Number of transmit buffer descriptors selection |

| ISDE Property | Value | Description |
|-----------------------------|--|---------------------------------------|
| Ethernet Interrupt Priority | Between 3 is reasonable. Priority 0 (highest), Priority 1:2, Priority 3 (CM4: valid, CM0+: lowest- not valid if using ThreadX), Priority 4:14 (CM4: valid, CM0+: invalid), Priority 15 (CM4 lowest - not valid if using ThreadX, CM0+: invalid) Default: Disabled | Ethernet interrupt priority selection |
| Name | g_sf_el_nx | Module name |
| Channel | 1 | Channel selection |
| Callback | NULL | Callback selection |

Note: The example settings and defaults are for a project using the S7G2 Synergy MCU Group. Other MCUs may have different default values and available configuration settings.

5.2 NetX BSD Support Module Clock Configuration

The ETHERC peripheral module uses PCLKA as its clock source. The PCLKA frequency is set by using the SSP configurator **Clock** tab prior to a build, or by using the CGC Interface at run-time.

5.3 NetX BSD Support Module Pin Configuration

The ETHERC peripheral module uses pins on the MCU to communicate to external devices. I/O pins must be selected and configured as required by the external device. The following table illustrates the method for selecting the pins within the SSP configuration window and the subsequent table illustrates an example selection for the I²C pins.

Note: The operation mode selection determines what peripheral signals are available and thus what MCU pins are required.

Table 9. Pin Selection for the ETHERC Module

| Resource | ISDE Tab | Pin selection Sequence |
|----------|----------|---|
| ETHERC | Pins | Select Peripherals > Connectivity:ETHERC > ETHERC1.RMII |

Note: The selection sequence assumes ETHERC1 is the desired hardware target for the driver.

Table 10. Pin Configuration Settings for the ETHERC1

| Pin Configuration Property | Value | Description |
|----------------------------|---|---|
| Operation Mode | Disabled, Custom, RMII (Default: Disabled) | Select RMII as the Operation Mode for ETHERC1 |
| Pin Group Selection | Mixed, _A only (Default: _A only) | Pin group selection |
| REF50CK | P701 | REF50CK Pin |
| TXD0 | P700 | TXD0 Pin |
| TXD1 | P406 | TXD1 Pin |
| TXD_EN | P405 | TXD_EN Pin |
| RXD0 | P702 | RXD0 Pin |
| RXD1 | P703 | RXD1 Pin |
| RX_ER | P704 | RX_ER Pin |
| CRS_DV | P705 | CRS_DV Pin |
| MDC | P403 | MDC Pin |
| MDIO | P404 | MDIO Pin |

Note: The example settings are for a project using the S7G2 Synergy MCU Group and the SK-S7G2 Kit. Other Synergy Kits and other Synergy MCUs may have different available pin configuration settings.

6. Using the NetX BSD Support Module in an Application

The following example assumes the NetX BSD Support module is TCP/UDP enabled and goes through the typical steps in using the NetX BSD Support module in an application:

NetX BSD Client:

1. Poll the `nx_ip_status_check` API for when the IP instance has a valid IP address.
2. Create a socket using the `socket` API.
3. Create `sockaddr_in` for client and server defining IP address and port for client and server.
4. Bind to a local source port using the `bind` API.
5. Connect to the server using the `connect` API.
6. Obtain connection information using `getpeername`, `getsockname` services [Optional]
7. Send a packet to the Server using the `send` API.
8. Receive a packet using the `recv` API.
9. Close the socket using the `soc_close` API.

NetX BSD Server:

1. Poll the `nx_ip_status_check` API for when the IP instance has a valid IP address.
2. Create a master socket using the `socket` API.
3. Create `sockaddr_in` defining IP address and port for server.
4. Bind the socket to a port using the `bind` API.
5. Assign a local source port to listen for client requests using the `listen` API.
6. Check for socket requests (read, write, exception) using the `select` API.
7. Accept client requests and hand off the connection to a secondary socket using the `accept` API.
8. Receive packets using the `recv` API.
9. Send packets using the `send` API.
10. Close the socket using the `soc_close` API.

These common steps are illustrated in a typical operational flow diagram in the following figures:

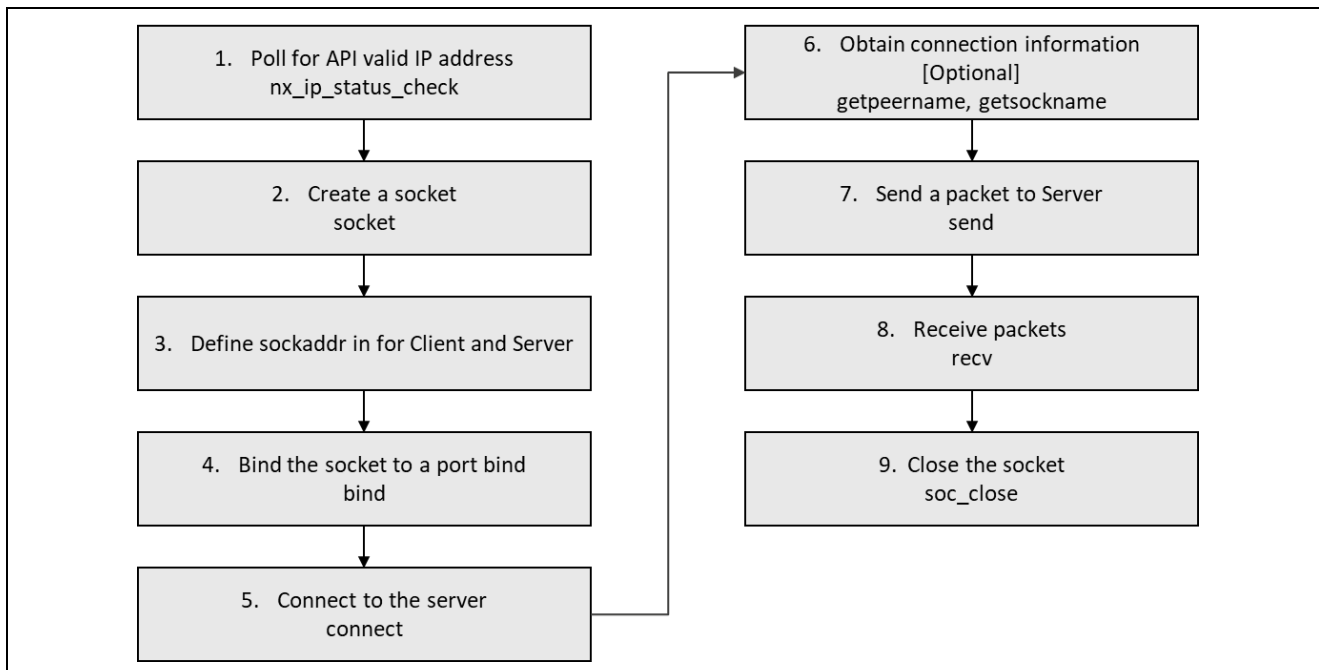


Figure 3. NetX BSD TCP Client Support Module

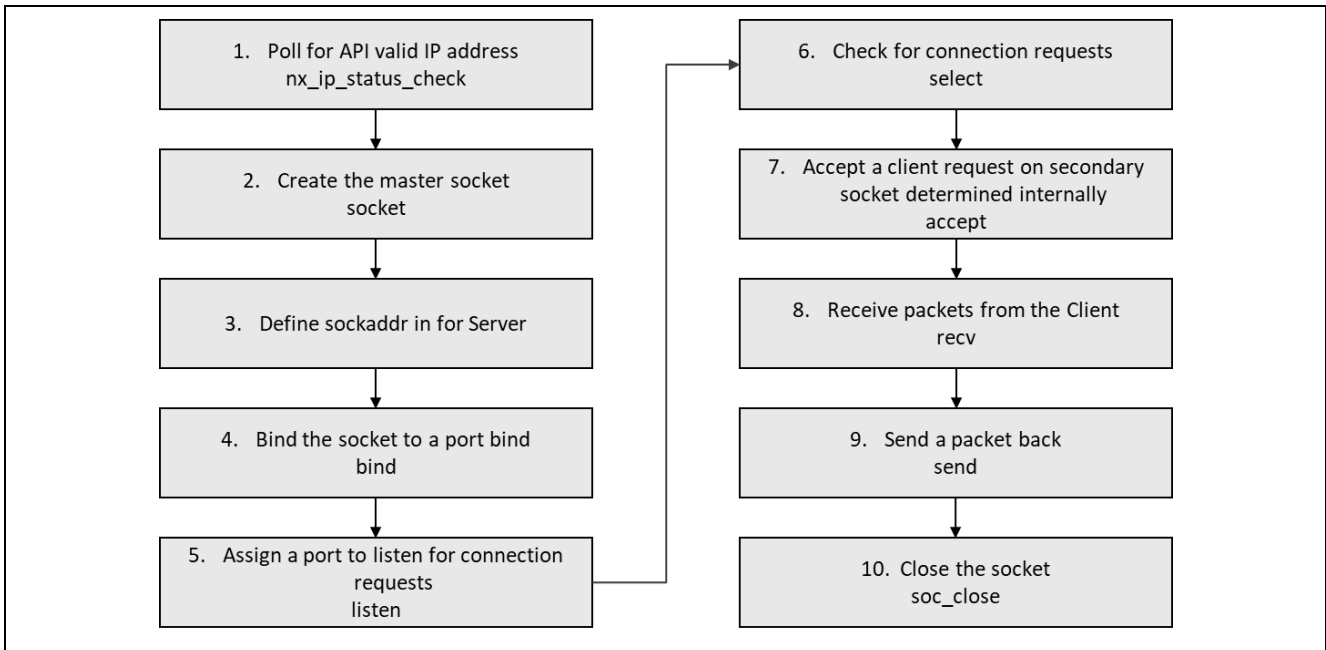


Figure 4. NetX BSD TCP Server Support Module

NetX Duo BSD Client

1. Poll the `nx_ip_status_check` API for when the IP instance has a valid IP address.
2. Create a socket using the `socket` API.
3. Create `sockaddr_in` (or `sockaddr_in6` for IPv6) defining IP (or IPv6) address and port for client and server.
4. Bind to a local source port using the `bind` API.
5. Connect to the server using the `connect` API.
6. Obtain connection information using `getpeername`, `getsockname` services [Optional]
7. Send a packet using the `send` API.
8. Receive packets using the `recv` and `send` API.
9. Close the socket using the `soc_close` API.

NetX Duo BSD Server

1. Poll the `nx_ip_status_check` API for when the IP instance has a valid IP address.
2. Create a master socket using the `socket` API.
3. Create `sockaddr_in` (or `sockaddr_in6` for IPv6) defining IP (or IPv6) address and port for server Set socket options using `ioctl`, `setsockopt` API
4. Bind the socket to a port using the `bind` API.
5. Assign a local source port to listen for client requests using the `listen` API.
6. Check for socket requests (read, write, exception) using the `select` API.
7. Accept client requests and hand off the connection to a secondary socket using the `accept` API.
8. Receive packets using the `recv` API.
9. Send packets using the `send` API.
10. Close the socket using the `soc_close` API

These common steps are illustrated in a typical operational flow diagram in the following figures:

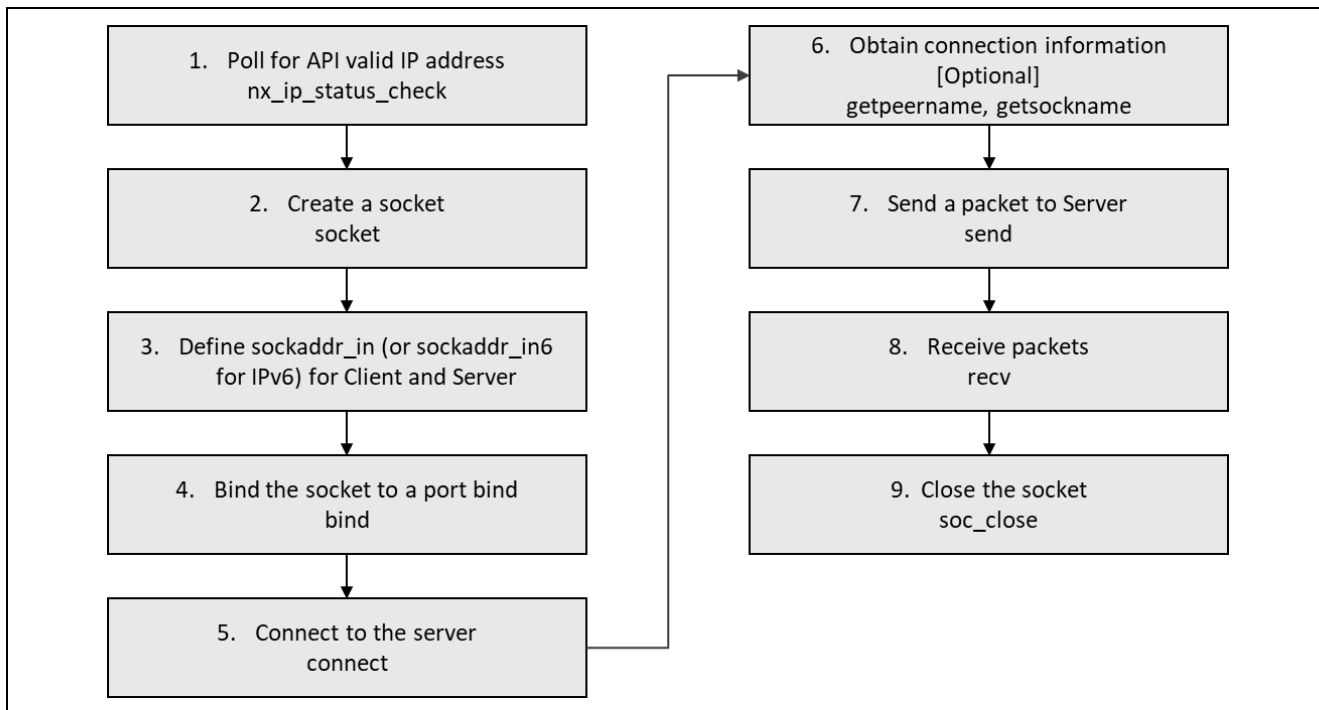


Figure 5. NetX Duo BSD TCP Client Support Module

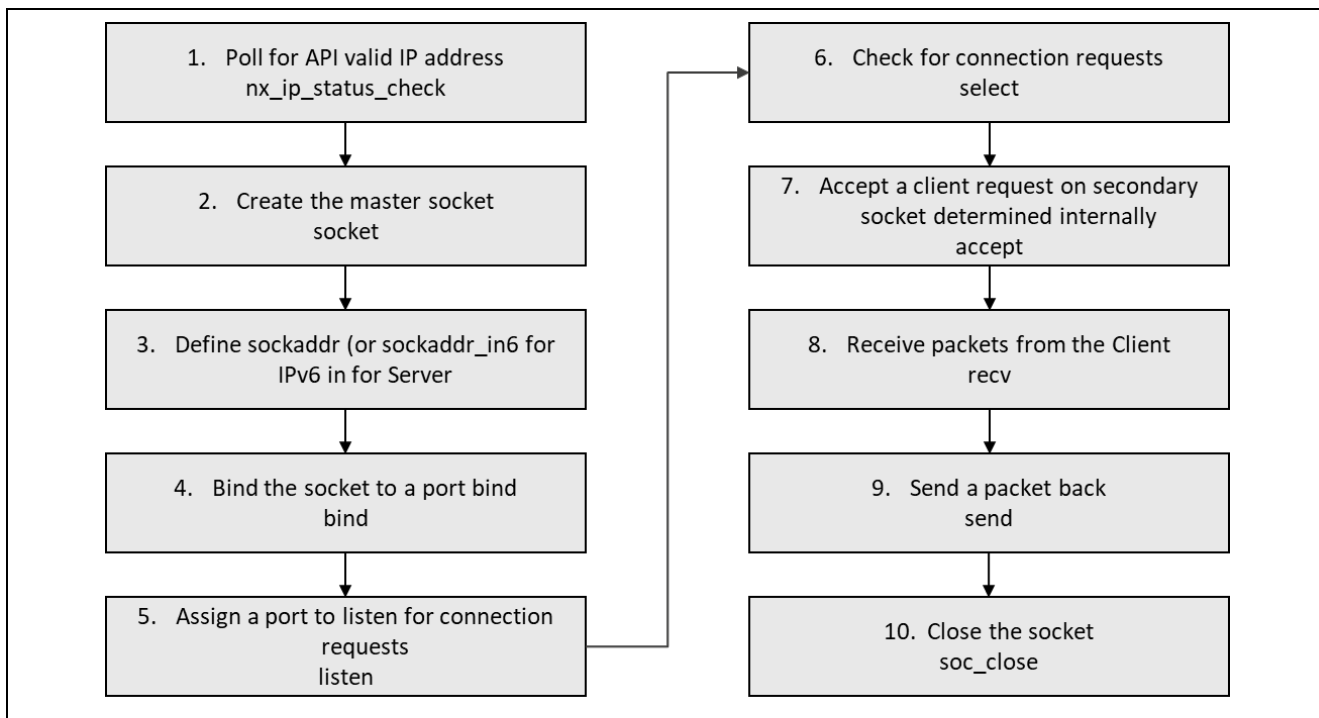


Figure 6. NetX Duo BSD TCP Server Support Module

7. The NetX BSD Support Module Application Project

This application project demonstrates the steps in a full design. The project can be found using the link provided in the Reference section at the end of this document. You may want to import and open the application project within the ISDE and view the configuration settings for the NetX BSD Support module.

The application project demonstrates the typical use of the NetX BSD Support module APIs. It sets up a BSD TCP Client socket on the board for connecting to a PC running a server socket Perl script, generating connection information, and sending and receiving a packet with the server before disconnecting and closing the socket. Then it sets up a BSD TCP Server socket on the board for accepting TCP Client connection

requests, exchanging packets, and releasing the accepting (secondary) socket after the client disconnects back to the pool of available sockets to accept connections.

Table 11. Software and Hardware Resources Used by the Application Project

| Resource | Revision | Description |
|----------------------------|-----------------|---|
| e2 studio | 7.3.0 or later | Integrated Solution Development Environment |
| SSP | 1.6.0 or later | Synergy Software Platform |
| IAR EW for Renesas Synergy | 8.23.3 or later | IAR Embedded Workbench for Renesas Synergy |
| SSC | 7.3.0 or later | Synergy Standalone Configurator |
| SK-S7G2 | v3.0 to v3.3 | Starter Kit |
| ActivePerl | 5.24.2 or later | Perl |

A simple flow diagram of the application project client and server functions are given in the figures below:

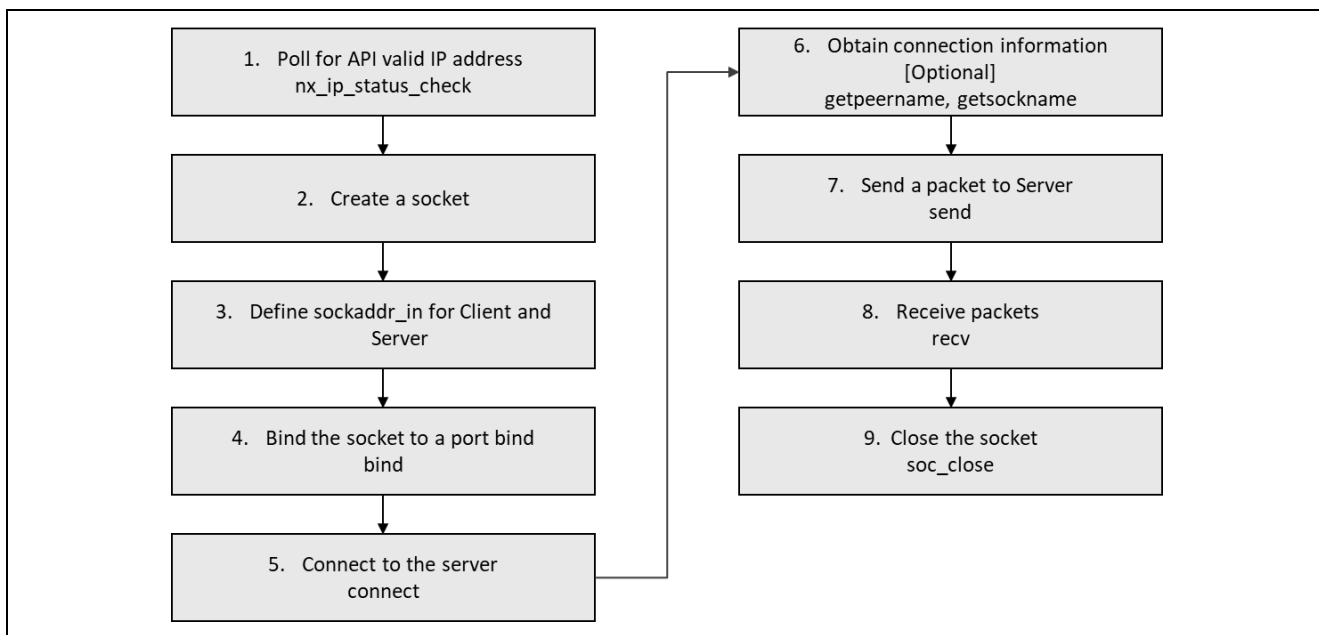


Figure 7. NetX BSD Support Module Application Project for TCP Client Flow Diagram

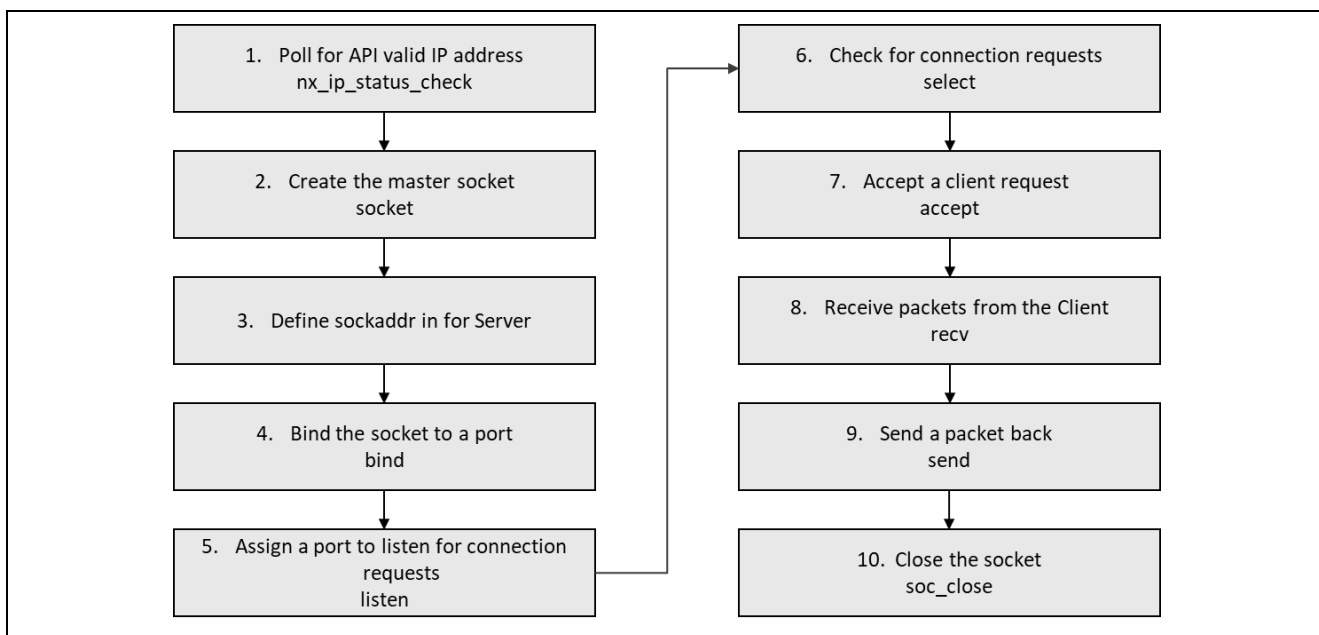


Figure 8. NetX BSD Support Module Application Project for TCP Server Flow Diagram

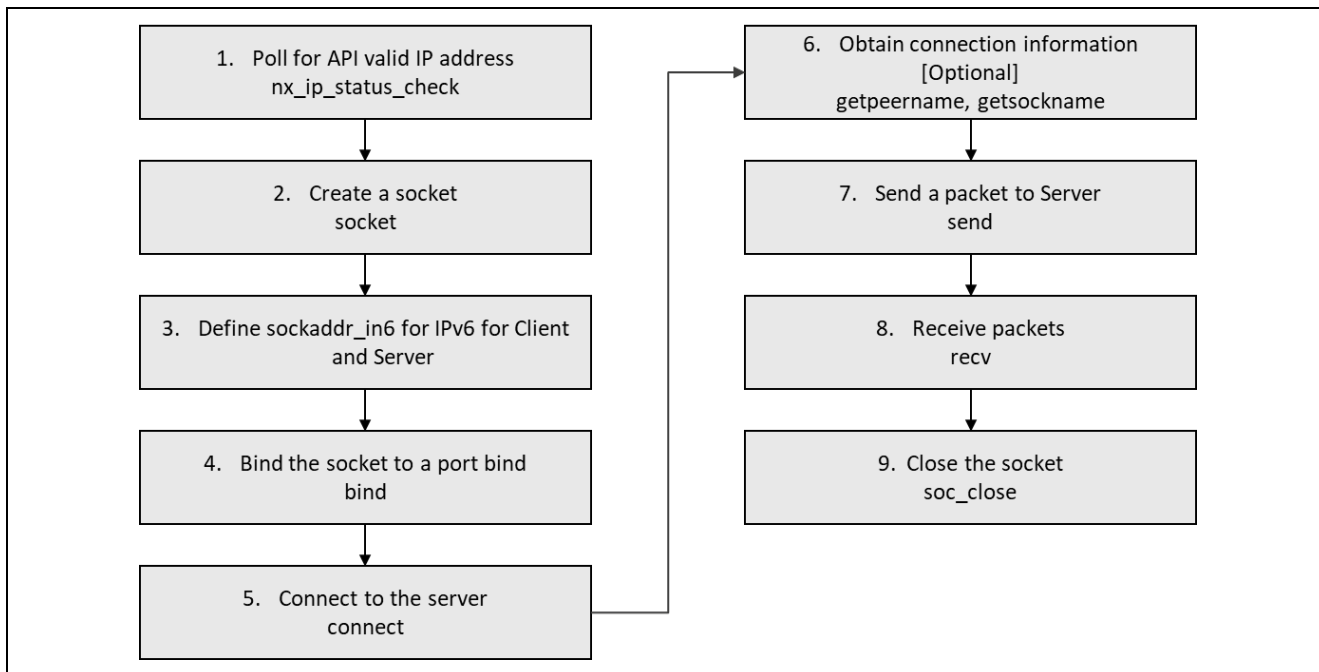


Figure 9. NetX Duo BSD TCP IPv6 Client

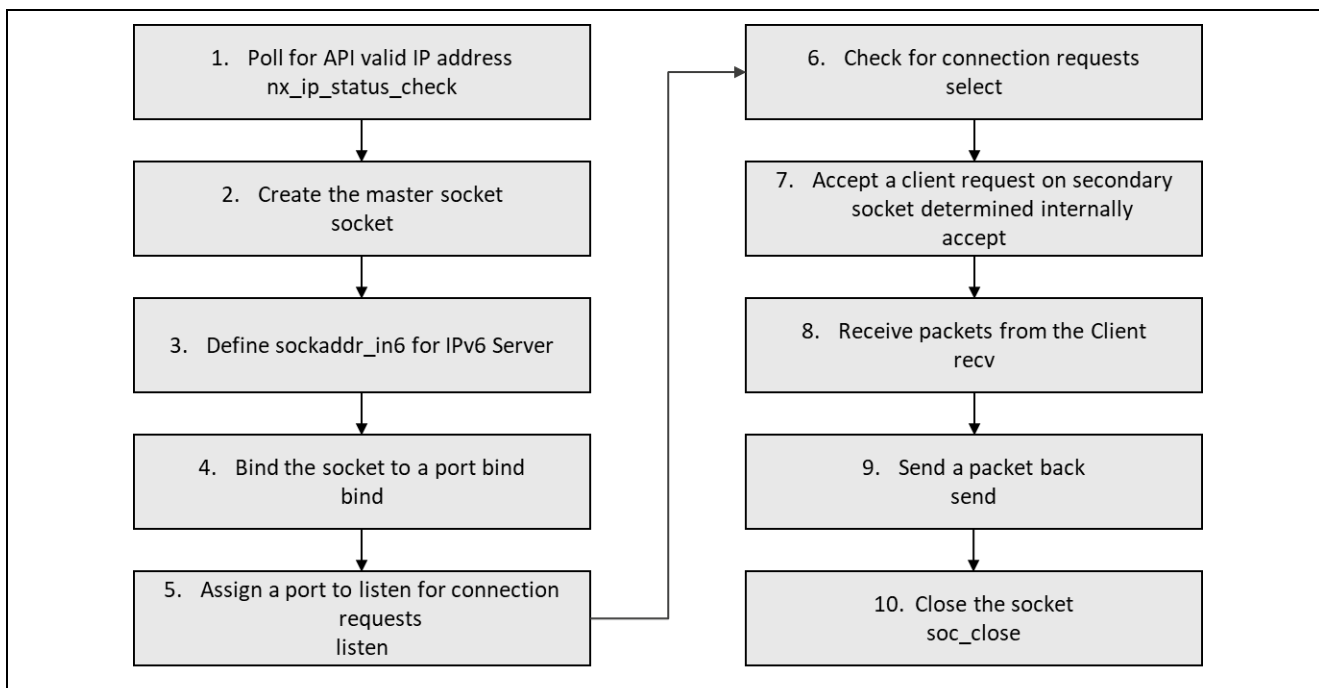


Figure 10. NetX Duo BSD TCP IPv6 Server

The `BSD_Server_Client_TCP_MG.c` file (and `BSD_Server_Client_TCP_MG.h`) is located in the project `src` folder once it has been imported into the ISDE. You can open this file within the ISDE and follow along with the description provided to help identify key uses of APIs.

The two main functions in the `BSD_Server_Client_TCP_MG.c` file are `run_bsd_server_session()` and `run_bsd_client_session()`. If `SEMI_HOSTING` is defined in `BSD_Server_Client_TCP_MG.h`, the output of the project is directed to the Renesas Debug Virtual console in the e² studio. The progress each function makes running a TCP session can also be viewed using a packet trace capture such as **tcpdump** or **Wireshark** running on any PC located on the same network. Note that since TCP packets in this application are unicast and only visible to the hosts on the network specified by the BSD local address and peer address.

Two Perl scripts are included in this package when run on a PC, will create a socket and communicate with the board running the example project. These scripts are:

socket_tcp_client_moduleguide.pl

socket_tcp_server_moduleguide.pl

NetX BSD TCP Server and Client Session Set up

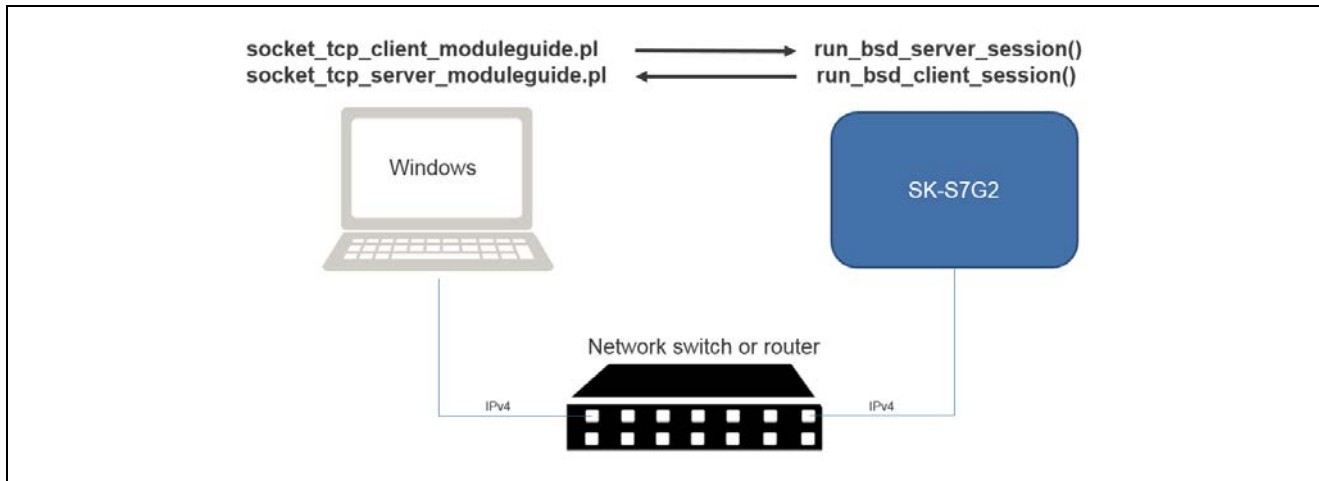


Figure 11. Physical setup

1. Connect the SK-S7G2 board and a Windows PC to a network switch or router.
2. Connect the ethernet RJ45 cable to J11 of SK-S7G2 board and the other side to switch or router.
3. Connect Micro USB cable to J19 and the other end to PC.
4. Connect another ethernet RJ45 cable from your PC and the other end to the same switch or router your using for SK-S7G2 board.
5. Run ipconfig in a command shell on the PC to determine the IP address of its wired Ethernet adapter.

```

C:\>ipconfig

Windows IP Configuration

Wireless LAN adapter Wireless Network Connection 2:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix . . :

Wireless LAN adapter Wireless Network Connection:

    Connection-specific DNS Suffix . . :
    Link-local IPv6 Address . . . . . : fe80::581a:9c8c:f42:667f%12
    IPv4 Address. . . . . : 143.103.201.59
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 143.103.201.254

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix . . :
    Link-local IPv6 Address . . . . . : fe80::55ae:46da:bc08:7da5%11
    IPv4 Address. . . . . : 143.103.4.172
    Subnet Mask . . . . . : 255.255.255.128
    Default Gateway . . . . . : 143.103.4.254

Tunnel adapter isatap {8691811C-E81E-48D4-8D61-B00D404CDE59}:
  
```

6. Update the constant ECHO_SERVER_IP_ADDRESS in the file BSD_Server_Client_TCP_MG.h with this IP address.

```

/* Define the client and server ports. For the server session, CLIENT_PORT has no effect.
*/
#define SERVER_PORT 87
#define CLIENT_PORT 77

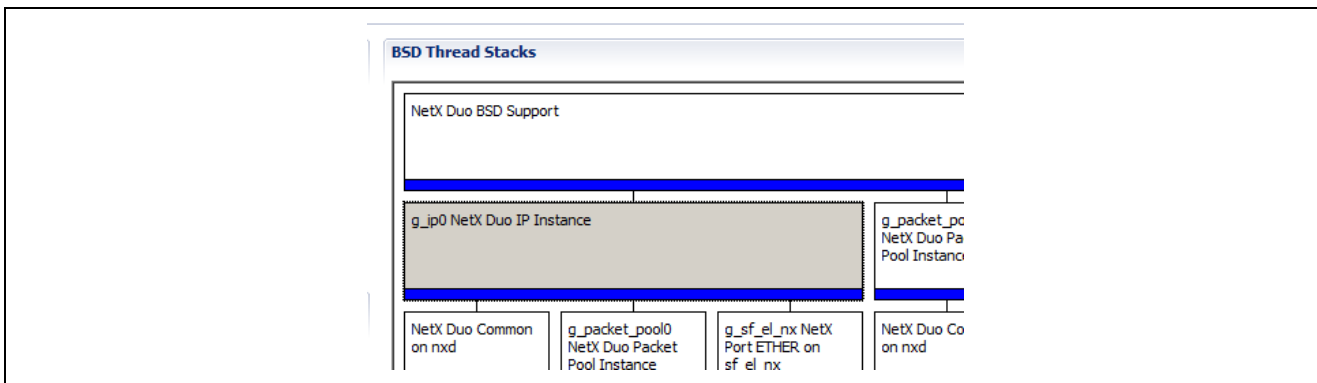
#define ECHO_SERVER_IP_ADDRESS IP_ADDRESS(143,103,4,172) /* IP address of your host system */

/* To send IPv6 packets, define DUO_IPV6. This is only available in NetX Duo
 * BSD Support.
#define DUO_IPV6
  
```


- Set the IP address of the board to an address that is on the same network as the PC but that is not taken. To determine if it is not taken run `arp -a` in a command shell on the PC. Alternatively you can ping the IP address you would like to assign the S7G2 and if nobody responds, you can use that IP address, at least for the time being.

```
C:\>arp -a

Interface: 143.103.4.172 --- 0xb
Internet Address      Physical Address      Type
143.103.4.132        50-7b-9d-fd-c8-47    dynamic
143.103.4.133        68-f7-28-fe-63-5b    dynamic
143.103.4.143        68-f7-28-69-c5-9b    dynamic
143.103.4.149        68-f7-28-7c-2c-4c    dynamic
143.103.4.168        00-15-99-75-13-69    dynamic
143.103.4.174        68-f7-28-7c-03-14    dynamic
143.103.4.254        00-00-0c-07-ac-21    dynamic
143.103.4.255        ff-ff-ff-ff-ff-ff    static
224.0.0.2            01-00-5e-00-00-02    static
```



| Property | Value |
|---|----------------------------------|
| Module g_ip0 NetX Duo IP Instance | |
| Name | g_ip0 |
| IPv4 Address (use commas for separation) | 143,103,4,173 |
| Subnet Mask (use commas for separation) | 255,255,255,0 |
| IPv6 Global Address (use commas for separation) | 0x2001,0xdb8,0x0,0x0,0x0,0x0,0x0 |
| True Link Local Address (use commas for separation. All zeros mean use MAC address) | 0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0 |

- Update the two Perl scripts with these two IP addresses using **Text Editor**
`socket_tcp_client_moduleguide.pl`:

```
#!/usr/bin/perl
# Create a client socket to make TCP client requests
use Socket;
my ($proto, $data, $message);

use IO::Socket::INET;

print "Client socket being created...\n";
$sock = IO::Socket::INET->new(
    LocalAddr => '143.103.4.172',
    LocalPort => '17',
    PeerAddr => '143.103.4.173',
    PeerPort => '87',
    Proto => 'tcp')
    or die "Can't create socket : $!";
```

`socket_tcp_server_moduleguide.pl`:

```

# Create a server socket to make TCP client requests
use Socket;
use IO::Socket::INET;
my $packet_count;
my $len;
my $recv_message;

$|= 1;

print "Server socket being created...\n";
my $sock = new IO::Socket::INET(
    LocalAddr => '143.103.4.172',
    LocalPort => '87',
    Proto => 'tcp',
    Listen => 1,
    Reuse => 1)
or die "Can't create socket : $!";
    
```

The auto-generated file `bsd_thread_entry.c` will call `run_bsd_client_session()` if `RUN_BSD_CLIENT` is defined in `BSD_Server_Client_TCP_MG.h`; if not defined, it will run `run_bsd_server_session()`.

- Define `DUO_IPV6` to run BSD over IPv6 (perl scripts not provided). Note that IPv6 is not supported in the NetX BSD.
- If `DUO_IPV6` is defined, and you are running the BSD Client socket session (`RUN_BSD_CLIENT` is defined), set the `localAddr6` IPv6 address for the PC running the TCP server.
- Make sure the *IPv4 Address* in the NetX IP Instance stack element in the project matches the device `sockaddr_in` addresses defined in `run_bsd_server_session` and `run_bsd_client_session` in `BSD_Server_Client_TCP_MG.c`.
- If you change `CLIENT_PORT` or `SERVER_PORT` (defined in `BSD_Server_Client_TCP_MG.h`) you will need to modify the perl scripts accordingly. By default they match and you need not change them.

Table 12. Configuration Settings for NetX IP Instance Support Module

| ISDE Property | Value | Description |
|--|---------------------|------------------------|
| Name | <code>g_ip0</code> | Module name |
| IPv4 Address (use commas for separation) | <device IP address> | IPv4 Address selection |

Table 13. Configuration Settings for the ThreadX Source Support Module

| ISDE Property | Value | Description |
|------------------------------------|-----------------------------|--|
| <code>TX_THREAD_EXTENSION_0</code> | <code>int bsd_errno;</code> | Enables error information in the thread context to be retrieved by the BSD support instance. |

Table 14. Configuration Settings for the NetX Source Support Module

| ISDE Property | Value | Description |
|-------------------------|---------|--|
| Extended Notify Support | Enabled | Enables callbacks in NetX to notify the BSD instance of certain events such as connection established and packet received. Required for NetX BSD Support projects. |

Table 15. NetX Port ETHER Configuration Settings for the Application Project

| ISDE Property | Value Set |
|---------------------------------|---------------------------------------|
| Parameter Checking | BSP, Enabled, Disabled (Default: BSP) |
| Channel 0 Phy Reset Pin | <code>IOPORT_PORT_09_PIN_03</code> |
| Channel 0 MAC Address High Bits | <code>0x00002E09</code> |
| Channel 0 MAC Address Low Bits | <code>0x0A0076C7</code> |
| Channel 1 Phy Reset Pin | <code>IOPORT_PORT_08_PIN_06</code> |
| Channel 1 MAC Address High Bits | <code>0x00002E09</code> |
| Channel 1 MAC Address Low Bits | <code>0x0A0076AB</code> |

| ISDE Property | Value Set |
|---------------------------------------|------------|
| Number of Receive Buffer Descriptors | 8 |
| Number of Transmit Buffer Descriptors | 32 |
| Ethernet Interrupt Priority | Priority 2 |
| Name | g_sf_el_nx |
| Channel | 1 |
| Callback | NULL |

- Set the **Extended Notify Support** to **enabled in the NetX Source component**. This is required for the NetX library to support BSD.
- Set the value of `TX_THREAD_EXTENSION_0`, `TX_THREAD_EXTENSION_1`, or `TX_THREAD_EXTENSION_2` properties to `int bsd_errno`; in the ThreadX source component.
- Your system may require `_POSIX_SOURCE` to be defined in the preprocessors list in the project compiler: **Right click** on the project in the Synergy Configuration view, choose **Properties -> C/C++ Build -> Setting -> Cross ARM C Compiler -> Preprocessor**. Click on the **+** icon in the **Defined symbols** pane, and type `_POSIX_SOURCE`.
- Click on **Generate Project** and then **build the project**.

8. Customizing the NetX BSD Support Module for a Target Application

There are numerous attributes that can be applied to BSD sockets to customize a socket for the application purpose. Some of the typical attributes to fine tune an application are provided as follows:

Non-blocking Sockets

BSD sockets can be set to non-blocking connect, accept, send/sendto, and recv/recvfrom calls have a `NX_NO_WAIT` option (**timeout = 0**). To make socket calls non-blocking, use the `ioctl` and `fcntl` API as follows on a previously created socket with descriptor `sockfd`:

```
fcntl(sockfd, F_SETFL, O_NONBLOCK)
```

```
int arg = NX_TRUE;
ioctl(sockfd, FIONBIO, &arg)
```

Note that internally NetX and NetX BSD Support use an internal timeout.

Setting socket options

Use `setsockopt` to set socket options. A common option is the reuse address option. To enable the reuse address (the default in NetX) set the value to 1. To disable, set the value to 0.

```
INT reuseaddr = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR &reuseaddr, sizeof(INT));
```

Two other socket options of interest are the `SO_SEDTIMEO` and `SO_RCVTIMEO` options. Respectively these set the timeout option on the `send` and `recv` calls. They set using the `setsockopt` API with the `SOL_SOCKET` option level. The input is a struct `timeval` data type:

```
Struct timeval time_0;
/* Set the timeout value to 50 ticks. The conversion of microseconds to
ticks is handled internally. */
time_0.tv_sec = 0;
time_0.tv_usec = 50 * NX_MICROSECOND_PER_CPU_TICK

setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &time_0, sizeof(time_0));
```

Getting socket options

Use the `getsockopt` to obtain information relevant to socket operations. A very common option is the `SO_ERROR` option.

```
INT error, len;
```

```
len = sizeof(error);
```

```
getsockopt(sock_tcp_client, SOL_SOCKET, SO_ERROR, &error, &len);
```

where the value of `error` is defined in `nx_bsd.h` for **NetX BSD Support**, and `nxd_bsd.h` for **NetX Duo BSD Support**.

errno

Another way to check for an error is the **errno macro**. If a connect or accept call on a non-blocking socket returns a non-successful result, check for indication of an **in-progress connection/accept**. This is not a failed connection. NetX returns this for non-blocking sockets:

```
if (errno == EINPROGRESS)
```

If any socket call returns a non-successful result, check if the connection is terminated. Not an error per se. The other side has closed the connection.

```
if (errno != ENOTCONN)
```

The difference between `getsockopt` with the `SO_ERROR` option and `errno`, is that for `getsockopt` the error is in the context of the socket and is set by BSD directly. For `errno`, the error is in the context of the BSD application thread and the **error** status is derived from the underlying NetX function call.

9. Running the NetX BSD Support Module Application Project

To run the **NetX BSD Support** module application project and to see it executed on a target kit, you can import it into your ISDE, compile, and run debug. Set up your hardware according to Figure 5 and refer to the *Synergy Project Import Guide* (`r11an0023eu0121-synergy-ssp-import-guide.pdf`, included in this package) for instructions on importing the project into e² studio or IAR Embedded Workbench® for Renesas Synergy™ and build/run the application.

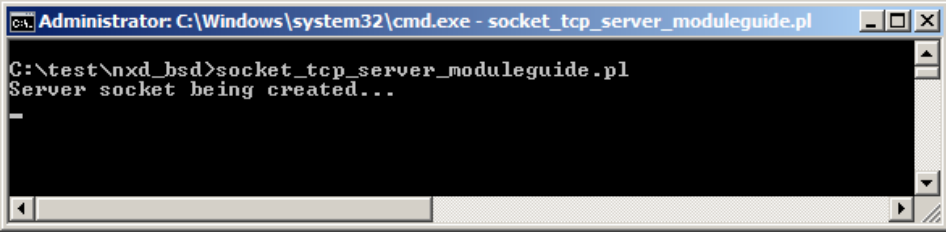
Running the TCP Client Session

A TCP Server must already be running on the network before calling `bsd_run_client_session()`. The project runs best if the TCP server and BSD TCP Client run on a local network, as opposed to through a switch to a larger network. `bsd_run_client_session` creates a TCP socket using the `socket` API, binds to a local port `CLIENT_PORT`/IPv4 address by calling the `bind` API, and connects to the TCP Server specified in the `echoServAddr` address instance by calling the `connect` API.

Note: The application should call `bind` explicitly before calling `connect()`. `connect()` will call `bind()` if it detects the socket is not bound. However, there is a bug in `connect()`, fixed in SSP releases following 1.5.0 such that the socket is not assigned a valid network interface.

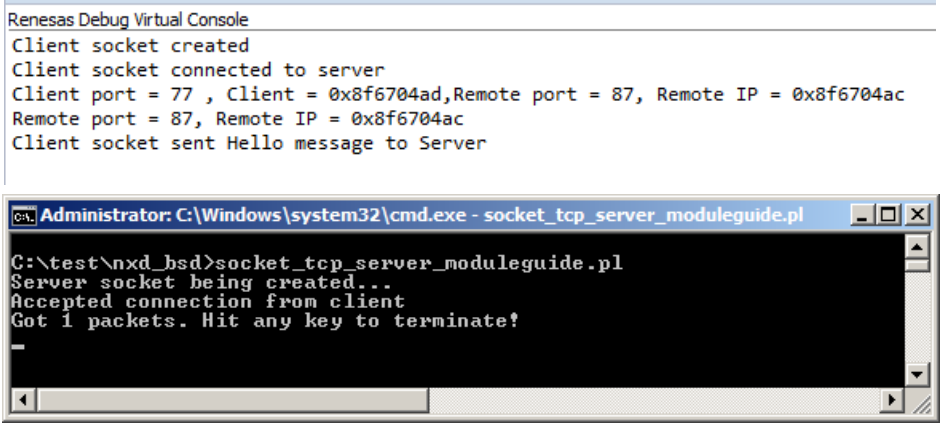
Once the connection succeeds, the application can check the server connection information by calling the `getpeername` API, and its own connection port and IP address. Then it sends a packet and waits to receive a packet, before closing (disconnects and closes internally) the socket by calling the `soc_close` API.

1. Open a command shell in Windows and run the Perl script `socket_tcp_server_moduleguide.pl`.



```
Administrator: C:\Windows\system32\cmd.exe - socket_tcp_server_moduleguide.pl
C:\test\nxd_bsd>socket_tcp_server_moduleguide.pl
Server socket being created...
```

2. Be sure the constant `RUN_BSD_CLIENT` is defined in the file `BSD_Server_Client_TCP_MG.h`.
3. Build the project and run it on the SK-S7G2 board with the virtual debugger window open in the tool.



```
Renesas Debug Virtual Console
Client socket created
Client socket connected to server
Client port = 77, Client = 0x8f6704ad, Remote port = 87, Remote IP = 0x8f6704ac
Remote port = 87, Remote IP = 0x8f6704ac
Client socket sent Hello message to Server

Administrator: C:\Windows\system32\cmd.exe - socket_tcp_server_moduleguide.pl
C:\test\nxd_bsd>socket_tcp_server_moduleguide.pl
Server socket being created...
Accepted connection from client
Got 1 packets. Hit any key to terminate!
```

4. A properly running example will display the output shown above.

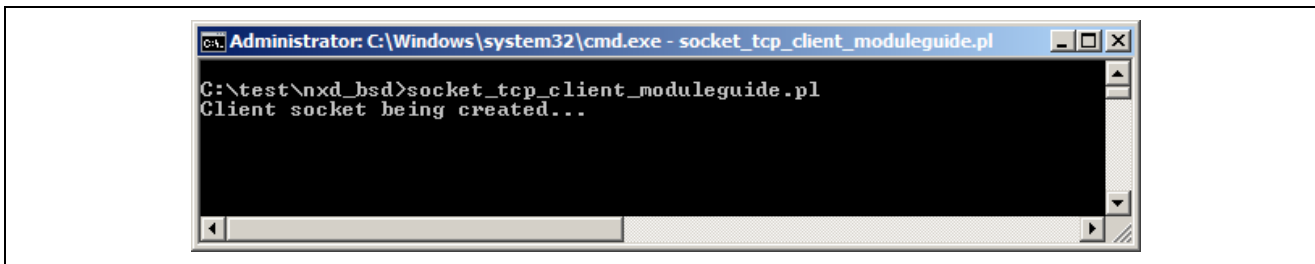
Running the TCP Server Session

The application checks that the IP address is resolved and NetX is ready to perform API services. A master TCP Server socket is created with the socket call. It then defines a `sockaddr_in` for its IP address and port. For IPv6 BSD connections in **NetX Duo BSD**, it defines a `sockaddr_in6` for the IPv6 address and port*. It binds to the port it will listen on. Then the socket is set to listen on the specified server port. It calls `select` until it gets a Client connection request. After starting the session, start a TCP Client on the local network to connect to this server. When the connection is successful, the server waits to receive a packet from the client. Then it returns the packet back. Then the server closes the socket (if the client hasn't already done so).

Note:

- Run the code and pause it.
- Run "socket_tcp_client_moduleguide.pl" file.
- Now keep the code running then we will see the output console.

1. Open a command shell in Windows and run the Perl script `socket_tcp_client_moduleguide.pl`.



2. Be sure the constant `RUN_BSD_CLIENT` is not defined in the file `BSD_Server_Client_TCP_MG.h` by commenting this line out:

```

/* By default, the BSD Server session runs. To run BSD as a TCP Client socket,
 * define this option.*/
//#define RUN_BSD_CLIENT
  
```

3. Build the project and run it on the SK-S7G2 board with the Renesas virtual debugger window open in e2 studio.

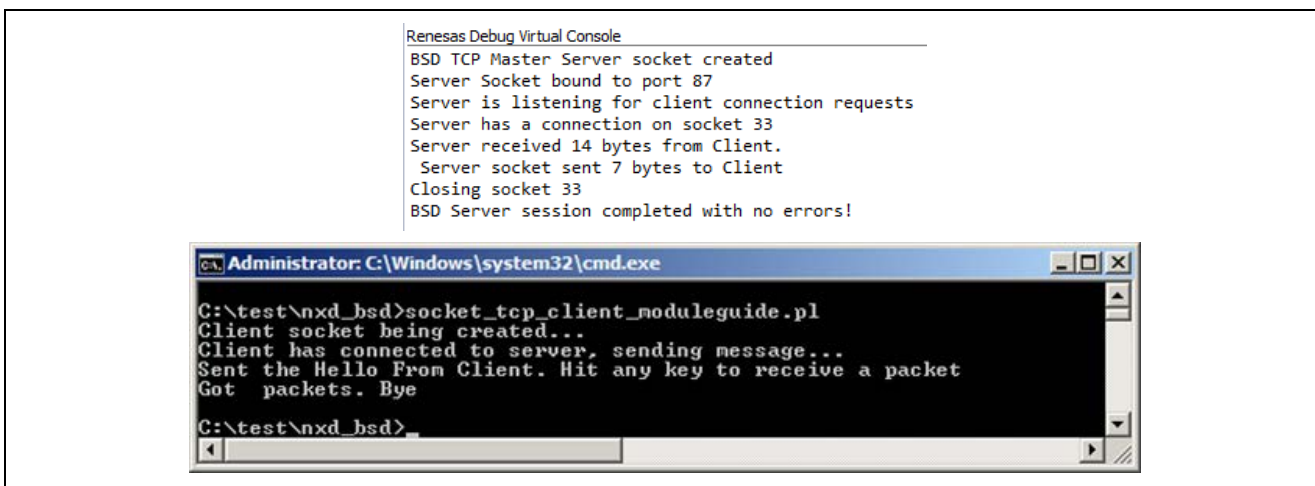


Figure 12. Example Output from NetX BSD Support Module Application Project

4. A properly running example will display the output shown above.

*Note that IP and IPv6 address information must be submitted in network byte order. When defining the `sockaddr_in` and `sockaddr_in6` data instances, the IP and IPv6 addresses are converted from host to network byte order using the `htonl` macro. The port is converted to a network byte order using the `htons` macro.

10. NetX BSD Support Module Conclusion

This module guide has provided all the background information needed to select, add, configure, and use the BSD module in an example project. Many of these steps were time consuming and error-prone activities in previous generations of embedded systems. The Renesas Synergy Platform makes these steps much less time consuming and removes common errors, like conflicting configuration settings or incorrect selection of lower-level drivers. The use of high-level APIs (as demonstrated in the application client and server projects) illustrates additional development-time savings by allowing work to begin at a high level and avoiding the time required in older development environments to use, or, in some cases, create, lower-level drivers.

For projects developed in BSD sockets, porting to a new TCP/UDP stack is potentially an arduous and lengthy process. The BSD interface to the NetX and NetX Duo stack greatly simplifies this task. Developing in the Synergy environment expedites most of the required steps, such as creating the IP instance, manually enabling basic services such as UDP and TCP support, creation of basic resources such as sockets, and setting up notification callbacks with NetX for socket events. The application is bundled with the NetX Port ETHER driver to eliminate the need to develop and debug low level drivers.

This module guide offers suggestions for further development of BSD projects, including discussion of more complex features like raw packet processing, using secondary interfaces, and setting socket options for non-blocking sockets and the TCP keep alive support.

11. NetX BSD Support Module Next Steps

After you have mastered a simple BSD module project you may want to consider more complex features. These include creating and using raw sockets (NetX Duo BSD only), joining and leaving IGMP multicast memberships using the `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP`, using the `setsockopt`.

The application project `run_bsd_server_session` function can be modified to continually check for new connection requests rather than terminating after a single connection. In this manner, the BSD TCP Server can handle multiple connections with TCP clients simultaneously which is the normal usage of BSD TCP Server applications. Or it can be simplified to remove the select statement logic and simply wait to accept a single connection request.

12. Reference Information

SSP User Manual: Available in html format in the SSP distribution package and as a pdf from the Renesas Synergy Gallery.

Links to all the most up-to-date NetX BSD Support module reference materials and resources are available on the Synergy Knowledge Base: <https://en-support.renesas.com/knowledgeBase/16977450>.

Website and Support

Visit the following vanity URLs to learn about key elements of the Synergy Platform, download components and related documentation, and get support.

| | |
|---------------------------------|--|
| Synergy Software | www.renesas.com/synergy/software |
| Synergy Software Package | www.renesas.com/synergy/ssp |
| Software add-ons | www.renesas.com/synergy/addons |
| Software glossary | www.renesas.com/synergy/softwareglossary |
| Development tools | www.renesas.com/synergy/tools |
| Synergy Hardware | www.renesas.com/synergy/hardware |
| Microcontrollers | www.renesas.com/synergy/mcus |
| MCU glossary | www.renesas.com/synergy/mcuglossary |
| Parametric search | www.renesas.com/synergy/parametric |
| Kits | www.renesas.com/synergy/kits |
| Synergy Solutions Gallery | www.renesas.com/synergy/solutionsgallery |
| Partner projects | www.renesas.com/synergy/partnerprojects |
| Application projects | www.renesas.com/synergy/applicationprojects |
| Self-service support resources: | |
| Documentation | www.renesas.com/synergy/docs |
| Knowledgebase | www.renesas.com/synergy/knowledgebase |
| Forums | www.renesas.com/synergy/forum |
| Training | www.renesas.com/synergy/training |
| Videos | www.renesas.com/synergy/videos |
| Chat and web ticket | www.renesas.com/synergy/resourcelibrary |

Revision History

| Rev. | Date | Description | |
|-------------|-------------|--------------------|----------------------------------|
| | | Page | Summary |
| 1.00 | Apr.18.17 | - | Initial Release |
| 1.01 | Dec.08.17 | - | Added NetX Duo, edit and release |
| 1.02 | May.02.19 | - | Updated for SSP v1.6.0 |

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
 - "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
 - "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.