EtherNet/IP Stack

(ESDK & EADK)

NetStaX™

Porting Guide

(For use with EtherNet/IP Stack version 5.2.0 and higher)

Pyramid Solutions, Inc.
30200 Telegraph Rd, Suite 440
Bingham Farms, Michigan 48025

Phone: 248-549-1200

Web: www.pyramidsolutions.com

PYRAMIDSOLUTIONS

VISIONARY SOLUTIONS ▲ EXCEPTIONAL RESULTS

Document Revision

Revision	Remarks	Date	Author
1.00	Initial Release	10/5/2006	MM
1.01	Updated for v3.7.0	8/24/2007	MM
1.02	Updated for v4.0.0	3/2/2009	MM
1.03	Updated for v4.1.0	8/5/2010	MM
1.04	Updated for v4.2.0	9/5/2010	MM
1.05	Updated for v4.3.0	8/28/2012	MM
1.06	Updated for v4.6.0	1/13/2016	MM
1.07	Updated for v4.8.0	1/29/2018	MM
1.08	Updated for v4.9.0	3/26/2019	PG
1.09	Updated for v5.1.0	6/10/2020	PG
1.10	Updated for v5.2.0	1/20/2021	PG

TABLE OF CONTENTS

Porting the EtherNet/IP Stack

This document describes how to port the ESDK or EADK Stack source code to a particular platform. The ESDK and EADK both rely on the same mechanism for porting, so switching between the two requires no additional effort.

All platform specific calls in the EtherNet/IP Stack code are separated out into a platform module comprised of a header files, Platform.h, and a collection of C files, Platform.c plus files for platform-specific CIP object functions. Complete platform modules for a variety of supported platforms and operating systems can be found in individual subdirectories under the \Src\Platform directory.

Windows Environments

Windows environments typically utilize the ESDK or EADK as a C++ DLL, and it can be built as such exactly as provided.

Windows

Appropriate workspace and project files (*EtIpScanner.sln/EtIpAdapter.sln* and *EtIpScanner.vcproj/EtIpAdapter.vcproj*) are provided in the Src directory to allow building the ESDK or EADK respectfully for Windows environments using the Microsoft Visual Studio 2015 development environment. The resulting .LIB and .DLL files along with the *EtIPScanner.h/EtIPAdapter.h* include file allow the library to be either implicitly (i.e. dynamically) or explicitly (i.e. statically) linked with the target application.

Loading the Library

The EtherNet/IP DLL can be linked to an application in one of two ways:

Static Linking

The example project provided statically links the ESDK or EADK DLL to the example application at build time by its inclusion of both the EtIPScanner.h/EtIPAdapter.h and EtIPScanner.lib/EtIPAdapter.lib files within the project.

Dynamic Linking

The ESDK or EADK DLL may also be dynamically linked with an application at run time via the usual Windows API function calls. For example:

```
m_hinstDll = ::LoadLibrary( "EtIPAdapter.dll" );
m_fpAdapterStart = (AdapterStart)::GetProcAddress( m_hinstDll, "EtIPAdapterStart" );
::FreeLibrary( m_hinstDll );
```


Other Environments

Non Windows environments typically either utilize the EtherNet/IP Stack as a linkable C library, or include and build its code directly with the client application code.

In either case, the Windows DLL C++ wrapper included with the EtherNet/IP Stack is not used. To eliminate it, do not include the following files in your project:

- EtIpScanner.cpp/EtIPAdapter.cpp and EtIPScanner.h/EtIPAdapter.h (for the ESDK and EADK respectively)
- StdAfx.cpp and StdAfx.h

Note however that in such environments there is a functional difference between the C API clientStart() (and *scannerStart()* for ESDK) function and C++ API EtIpScannerStart/EtIPAdapterStart() function. The *EtIPScannerStart()* and *EtIPAdapterStart()* function calls *clientStart()* and then creates a thread which continuously calls *clientMainTask()* at periodic (e.g. 1 msec) intervals. When utilizing the stack "C" API, those responsibilities must instead be handled by the client application itself.

Linux

An appropriate make file (*Esdk.mk/Eadk.mk*) as well as source code for a simple wrapper application (*EsdkDemo.c/EadkDemo.c*) are provided in the Src\Platform\Linux directory to allow building the ESDK or EADK for Linux using the Gnu C compiler. The make file contains relative paths to the Src directory.

The *Esdk/Eadk.mk* makefiles will build the stack as a static library. The *EsdkDemo/EadkDemo.mk* makefiles will build the sample application linking with the static library.

The Linux makefiles are 64/32-bit aware and will build the stack library and demo application with the appropriate 64 or 32-bit settings based on the platform on which the make is running.

Other Platforms

When porting the EtherNet/IP Stack to other platforms, the user will have to create platform files appropriate for the target hardware and OS utilized.

A set of platform template files are included in the Platform\Templates directory which provide functions prototypes, stubs and macros along with comments describing the required functionality and usage. A few other ported platforms are also included in the Platform directory.

For complete details on the platform requirements, see the comments in the template platform files. In general terms, the platform specific functionality which must be provided includes:

- **GetTickCount() implementation** – A function which provides the EtherNet/IP Stack with a way of calculating elapsed times (in milliseconds) in order to maintain the proper connection rates and determine when messages or connections should time out;
- **Sockets implementation** – Usually simple macros which address naming differences between various sockets implementations; Socket API is based on Berkeley sockets.
- **Low Level Network Data –** A few functions which provide the EtherNet/IP Stack with low level network data such as the Ethernet MAC ID and TCP/IP configuration settings for the TCP/IP and Ethernet Link Objects.
- **Mutex implementation** – A mutex is used to synchronize the execution of the EtherNet/IP Stack main thread and the application level thread that is using the stack's services.
- **Log file generation** - The user may enable log file generation functionality to have debug trace functionality.
- **"Big 12" Diagnostics** – If the EIP_BIG12 build option is included, more attributes are available to be populated in the Ethernet Link object and the platform must provide information about CPU utilization.
- **Platform specific handling for built in CIP Objects** – The CIP objects implemented by the stack interface with the platform to read and write attribute values and handle any specific Set handling. There are platform functions for the TCP/IP, Ethernet Link, QoS and File objects.
- **Non-Volatile configuration handling** – Any object attributes that must be maintained in non-volatile storage must be stored and retrieved by the platform.
- **QoS functionality (Optional)** – The EtherNet/IP Stack provided the QoS object through the EIP_QOS #define, but platform hooks need to be implemented for setting DSCP values and possibly adding IEEE 802.1D/Q frames.
- **Address Conflict Detection (ACD) (Optional)** – A function is provided through the EIP_ACD #define to inform the EtherNet/IP Stack if a duplicate IP address is detected. It is up to the platform to correctly implement the ACD algorithm as specified in the EtherNet/IP specification. The platform must also populate the TCP/IP object attributes related to ACD.
- **File object functionality (Optional)** – A interface to the platform's file system that will be used by the stack File object to create, delete, read and write files.

Additional Socket Implementation Notes

The biggest incompatibility with the EtherNet/IP Stack's socket implementation is the use of the platformSelect() call (typically mapped to select()). The platformSelect() call is used to determine when a socket has successfully connected (or failed with the exceptionset field) via platformConnect() (typically mapped to connect()). It is NOT used to determine if sockets have data to read or write.

If the platform doesn't support select() or the connect() call is non-blocking, a separate thread can be created to mimic the functionality in the EtherNet/IP Stack. Start by defining SYNCHRONOUS_CONNECTION (removes the platformSelect() and platformConnect() functionality in the EtherNet/IP Stack). Below is an example thread function implementing the connect() call without select(). Specific platform modifications should only need to replace or modify the connect() call, leaving the rest of the function.

```
void TCPConnectTask( void *pd )
{
        SESSION* pSession;
        SOCKET   lSocket;
        INT32    nSessionId;
        UINT32   lClientIPAddr;

        platformSleep(1);

        while (1)
        {
                platformWaitMutex(ghClientMutex, MUTEX_TIMEOUT);
                for( pSession = gSessions; pSession < gpnSessions; pSession++ )
                {
                        if ( pSession->lState == OpenSessionLogged )
                        {
                                DumpStr0(TRACE_LEVEL_NOTICE,
TRACE_TYPE_SESSION, 0, 0, "TCPConnectTask connect");
                                pSession->lState = OpenSessionWaitingForTCPConnection;

                                nSessionId = pSession->nSessionId;
                                lClientIPAddr = pSession->lClientIPAddr;

                                platformReleaseMutex(ghClientMutex);

                                lSocket = connect( lClientIPAddr, 0, htons(
ENCAP_SERVER_PORT ), CONNECT_TIMEOUT );

                        platformWaitMutex(ghClientMutex, MUTEX_TIMEOUT);
```



```
                    pSession = sessionGetBySessionId( nSessionId );

                    if ( pSession == NULL )
                       break;

                    if ( lSocket > 0 )
                    {
                            SetSocketRxBuffers( lSocket, 15 );
                            setsocketackbuffers( lSocket, 15 );

                            /* Save the info for the newly connected socket */
                            pSession->lSocket = lSocket;
                            pSession->lState = OpenSessionTCPConnectionEstablished;
                            DumpStr1(TRACE_LEVEL_NOTICE,
                    TRACE_TYPE_SESSION, pSession->lHostIPAddr, lClientIPAddr,
                    "Connect() succeeded with session Id 0x%x", nRet);

                    }
                    else
                    {
                            DumpStr1(TRACE_LEVEL_NOTICE,
                    TRACE_TYPE_SESSION, pSession->lHostIPAddr, lClientIPAddr,
                    "Connect() failed with error code %d", nRet);
                            sessionRemove( pSession, FALSE );
                    }

                    break;
            }
        }

        platformReleaseMutex(ghClientMutex);

        platformSleep(1);
        }
}
```


Ethernet/IP PlugFest Considerations

The Ethernet/IP Stack supports all CIP object attributes required by the Ethernet/IP Plugfest requirements. However, it is up to the platform to provide support for getting and setting those attribute values (and possibly saving values in NV storage). This includes ACD functionality. Not all platforms provide this capability, so while a device may run Ethernet/IP and pass conformance testing, it may not pass Plugfest requirements.

The Windows and Linux platform files provided allow EtherNet/IP to operate, but they do not support setting the (TCP/IP and Ethernet Link) object attributes necessary to pass all Plugfest requirements.

Memory Considerations

When porting the EtherNet/IP Stack to other platforms, both the size of the ultimate binary executable as well as the size of data allocated by the stack should be considered.

The size of the binary EtherNet/IP Stack executable image depends on the development tools used to produce it. In most cases, 96K should be sufficient to accommodate the ESDK or EADK binary executable.

All of the data (RAM) memory required by the EtherNet/IP Stack can be allocated at stack startup. This ensures that the stack will not run out of memory later. The maximum data memory (RAM) required by the EtherNet/IP Stack at run time can be dictated by modifying a variety of system constants. The constants that affect the largest chunks of memory include:

- **MAX_SESSIONS** - Maximum number of network peers to which the EtherNet/IP Stack may maintain *simultaneous* communications sessions. If the EtherNet/IP Stack has to originate messages to a particular peer and respond to messages from the same peer, 2 sessions will be allocated – one for sending requests and receiving responses and another for receiving requests and sending responses. Each potential session instance takes 52 bytes of data. The default MAX_SESSIONS value is 128.
- **MAX_CONNECTIONS** - Maximum number of *simultaneous* connections that can be opened by and to the EtherNet/IP Stack. Each potential connection instance takes 252 bytes of data. The default MAX_CONNECTIONS value is 128.
- **MAX_REQUESTS** - Maximum number of *simultaneous* outstanding unconnected requests that can be opened. A Request is outstanding from the moment the EtherNet/IP Stack receives the send request command from the application layer and the moment the application layer reads the response data. Each potential outstanding request takes 80 bytes of data. The default MAX_REQUESTS value is 100.
- **MAX_ASSEMBLY_SIZE** – The largest supported assembly size. With the addition of support for the LargeForwardOpen, this can get quite large. Default is 1502 bytes.
- **ASSEMBLY_SIZE** - The size of the input and the output assembly data areas to store the I/O data. The Default ASSEMBLY_SIZE value is (MAX_ASSEMBLY_SIZE*MAX_ASSEMBLIES).

- **MAX_ASSEMBLIES** – The maximum number of assemblies that the application can support. This corresponds to the number of ASSEMBLY structures allocated, which defaults to taking 532 bytes (mostly due to MAX_MEMBERS). The default MAX_ASSEMBLIES is 128.
- **MAX_MEMBERS** – The maximum number of members supported within a single assembly. This is what gives the ASSEMBLY structure most of it size through the EtIPAssemblyMemberConfig structure which is 8 bytes per member. If members are not supported by the application, MAX_MEMBERS can be set to 1. The default MAX_MEMBERS is 64.
- **MAX_HOST_ADDRESSES** – The maximum number of IP addresses that can be supported on the device. A TCPIP_INTERFACE_INSTANCE_ATTRIBUTES structure is allocated for each IP addresses supported at a size of approximately 220 bytes (depending on TCP/IP object functionality supported). The default MAX_HOST_ADDRESSES value is 16.
- **MEMORY_POOL_SIZE** - Dynamic data area used to store the actual request data and the variable length strings including connection paths, names, and tags. The Default MEMORY_POOL_SIZE value is 64K. Note that this only takes affect if EIP_STACK_MEMORY is #defined. If it is not defined, the platform's *malloc()* and *free()* functions will be used

Using the default values defined above, the total RAM required for a Windows target amounts to 554K (due mostly to default ASSEMBLY_SIZE). Note that the RAM requirements can be reduced dramatically by reducing one or more of the maximum values described above.

For example, for a low-to-medium level system with 32K of RAM available, the recommended maximum values are as follows:

- MAX_SESSIONS - 32,
- MAX_CONNECTIONS – 16,
- MAX_REQUESTS – 32,
- ASSEMBLY_SIZE – 4K,
- MEMORY_POOL_SIZE – 16K.
- MAX_HOST_ADDRESSES – 1

Big Endian & Little Endian Support

The ESDK and EADK support both "Big Endian" and "Little Endian" memory models. To enable big endian support, the EIP_BIG_ENDIAN macro should be *#defined* in platform.h or as a preprocessor definition. For little endian platforms, don't include the macro.

Do not define the EIP_BIG_ENDIAN macro for Windows and Windows CE targets since these operating systems utilize the little endian data format.

64-bit and 32-bit Support

Some portions of the ESDK and EADK handle memory and pointers such that the stack must be aware of whether the platform architecture is 64-bit or 32-bit. If the stack will be run on a 64-bit platform, the EIP_64BIT compiler directive must be defined when building the stack.

Porting Application Objects

The EtherNet/IP Stack provides some CIP objects that aren't necessary for the basic operation of EtherNet/IP functionality. They are considered optional objects by most Device Profiles. They can be included in a device by registering them through clientRegObjectsForClientProc(). Since they are usually optional for a device, it didn't make sense to require porting functionality that was specific to those objects. The API used in the objects reflects the external API used by any device specific (application) object.

Time Sync Object

The Time Sync Object is the CIP object that represents IEEE-1588 v2 time synchronization functionality. The platform file for the Time Sync object provides hooks to get/set attributes that affect the behavior of an IEEE-1588 v2 stack. An IEEE-1588 v2 stack is NOT included with the EtherNet/IP Stack.

Device Level Ring (DLR) Object

The DLR Object provides the configuration and status information interface for the DLR protocol. The DLR protocol is a layer 2 protocol that enables the use of an Ethernet ring topology. The platform file for the DLR Object provides hooks to get/set attributes that affect the behavior of the DLR protocol. Both supervisor and non-supervisor functionality is supported through the DLR_RING_SUPERVISOR #define. A supervisor supports much more functionality.

CIP Energy Objects

The CIP Energy Objects provide energy and power reporting for a device. The porting process involves registering for the attributes a device supports for the energy object(s) it supports. The registered functions within the application are then called when requests for the object attributes are received. In addition to the registered attributes, instances of the object need to be created as well. An example of a porting of the CIP Energy objects can be found in EIP0021.

Performance Considerations

Overall performance of the EtherNet/IP Stack depends on the following factors:

- UDP traffic load - number of Class1 packets per second handled by the EtherNet/IP Stack;
- TCP traffic load - number of UCMM and Class3 packets per second handled by the EtherNet/IP Stack;
- PC processor speed - The performance of the EtherNet/IP Stack increases proportionally to the CPU speed. For example, when upgrading to a processor with a 2x performance increase, you can expect stack performance to increase about twofold;
- Network speed - 100Mb/sec will provide better performance compared to the 10Mb/sec.
- Network utilization - To reduce network utilization you should allocate a separate network for the EtherNet/IP devices;
- Network type - Full-duplex will provide better performance and compatibility than half-duplex.

For optimal performance and consistency please make sure that all devices are using Ethernet speed of 100Mb/sec and full-duplex protocol. Use the Windows Control Panel Network applet on the PC running the EtherNet/IP Stack to switch to 100 Mb/sec and full-duplex.

Listed below, we have included some performance tests run against the ESDK. The test results provide a rough estimate of ESDK performance for similar PC configurations listed below:

The test configuration consisted of:

- ESDK Scanner;
- 65 1793- and 1794- series Flex I/O modules connected to 14 1794-AENT adapters;
- ControlLogix 5555;
- Signal Generator.

All devices were connected in a closed circuit full-duplex 100M network.
Two PC hardware configurations were used to run the ESDK:

- Pentium II 333 with 128MB RAM; and
- Pentium III 866 with 256 MB RAM.

Both PCs had the Windows 2000 operating system.

The tests started with the ESDK Scanner opening 65 Class1 connections to the Flex I/O modules. The Requested Packet Rate (RPI) was set to 10 msec for the Pentium III 866MHz PC and 20 msec for the Pentium II 333MHz PC. A signal generator was used to change the input of the Flex I/O module. The ESDK Scanner was responsible for detecting the input change and updating the output of another Flex I/O module accordingly. The ControlLogix controller recorded the delay between the input change by the signal generator and the output change by the ESDK Scanner.

Each test was run for 10 hours and consisted of 360,000 iterations. A cyclic connection type was used to provide an “ideal” maximum delay of 2 times the RPI. The “ideal” maximum delay is

the Consuming Rate + Producing Rate for the Cyclic connections and the Producing Rate for the Change Of State and Application Triggered connections. The difference between the recorded maximum delay and the "ideal" maximum delay is the maximum processing time (or maximum overhead) imposed by the network, the operating system and the ESDK.

The Pentium II 333MHz PC based test with a traffic load of 6500 UDP packets/sec, recorded a maximum processing time of 30 msec. The Pentium III 866MHz PC based test with a traffic load of 13000 UDP packets/sec recorded a maximum processing time of 20 msec.

Sales & Support Information

Product Technical Support

If you require product specific technical support, please contact Pyramid Solutions' Product Technical Support team as follows:

1) Send an Email to productsupport@pyramidsolutions.com

   This method is the fastest because it immediately reaches all support engineers and allows you to specify the specific product and question / issue. We suggest that you specify the product in the email subject e.g. "ESDK Support Request" and provide a detailed description of your question / issue in the body of the email. A product engineer will either respond by email or will call you to initiate a discussion.

2) Call for support

   248-549-1200 (Pyramid Solutions' Bingham Farms, MI USA office)

   When prompted for "Additional Options", Press 1, then when prompted for "Customer Support", Press 2, and when prompted for "BridgeWay and NetStaX Support", Press 1.

Note: You must have available built-in product support hours remaining or have purchased additional support services to receive technical support. Support will be provided based on resource availability between the hours of 9:00 am to 5:00 pm EST (Monday through Friday except for Holidays). If you require additional support or other services, please contact the sales number shown below.

Services & Product Sales Contact Information

Pyramid Solutions, Inc.
Headquarters
30200 Telegraph Road, Suite 440
Bingham Farms, Michigan 48025

Phone: (248) 549-1200
1-888-PYRASOL

FAX: (248) 549-1400

Web: www.pyramidsolutions.com

