WEINZIERL ENGINEERING GMBH
Jason Richards
84558 Tyrlaching
GERMANY
Phone +49 (0) 8623 / 987 98 - 03
Web: www.weinzierl.de

Development of complex KNX Devices

Abstract

The KNX World is constantly growing in terms of the number of members, products and installations. This is matched by the growth of the requirements of the devices. These requirements can be fulfilled with new KNX device model, System B, providing considerably more resources. New devices can also be enhanced by selecting a high-performance and scalable system such as Linux or WinCE. We outline a daemon for Linux with a RPC protocol, separating the standard KNX application into a stand-alone client process. This daemon connects to the KNX network with either a low cost solution for twisted pair or with a direct connection via KNXnet/IP. New development tools have also be developed to manage the development workflow and product database creation. We present an overview of the technology and components available to achieve high quality and cost effective products.

1 Complex KNX Devices

A Complex KNX Device has one or more of the following attributes:

Complex KNX Model: defines its complexity in the use of its KNX resources, such as the number of Group Objects (Communication Objects) and the number of ETS Parameters. A device that implements runtime Group Objects approaching or exceeding 255 can be considered complex.

Complex Application: defines its complexity outside of KNX and can be categorized by its demands on platform resources, including the use of Operating System primitives (for multitasking and synchronisation), its requirements for Flash and RAM, and also its use of external software components, such as a web server etc. Typically these applications do not fit within the constraints of a small embedded environment and profit from the use of shared or dynamic libraries and Linux or Windows CE. An example of a complex application is a KNX-aware VoIP Phone.

The choice of platform and development environment for a complex KNX application depends on a number of factors, such as the development skills, experience and knowledge available, extant products and platforms and the ability to obtain lower component costs due to higher volume purchasing etc. Equally important is the device and application type as outlined about, as they require varying system-level decisions depending on whether you have a complex KNX model, a complex application environment, or both. A complex KNX interface can benefit from the use of the System B Device Model (outlined in detail below in Section 3.2). System B enables application developers to utilise up to 10,000 (and more) Group Objects [1]. A complex application can benefit from the use of a high-performance and scalable Operation System, such as Linux or WinCE. For complex KNX interfaces coupled with demanding application requirements we recommend our System B KNX Stack for Linux [2].

Another important factor that will influence (and in some cases define) system-level KNX design is the targeted KNX media type, one of Twisted-Pair, IP, Powerline or RF. Various device models (i.e. System 7, System B) are supported for the various KNX media; the matrix is outlined below. The remainder of this paper will describe the available Device Models (in Section 2), Solutions for Linux and WinCE (in Section 3) and Specifying a Product Database (Section 4) for developing robust and cost-effective KNX devices of a complex nature.

2 Device Models

The system software for KNX, implemented by each device certified by the KNX Association, is generally called the “KNX Stack” [3]. When referring to a stack for KNX, it is common to consider only the core of every stack implementation, the communication layers of the OSI/ISO reference model. A KNX capable device however additionally requires a Device Model, which specifies the management profile for either System or Easy Mode. The choice of Device Model depends on a number of factors, not limited to the targeted bus medium and the resource constraints / requirements of the device. There are currently two Device Models available for complex KNX device development: System 7 and System B.

2.1 System 7

System 7 is currently the most widely adopted Device Model. Due to the balance between the complexity of its management functions and the size of its resources it is used in many new development projects. Up to 255 Group Objects and 254 Group Addresses are available, and the memory model supports up to 30kB for binary data / loadable code. The device management, based on properties, is extensive and does not require the use of fixed memory address. The memory layout is specified by the developer using Load Controls and it is possible to define Segments with varying functionality.

2.2 System B

System B is a new KNX Device Model which adopts many of the proven management procedures from System 7. Most notable is the extensive System Management via Interface Object Properties, which greatly obviates the need for direct memory access (DMA). ETS is still able to use DMA however for performance gain during the download procedure. Due to its re-locatable memory management, base address pointers to the tables have to be read which increases the complexity of the download procedure somewhat, although this complexity is mostly hidden within the ETS. The use of re-locatable memory management in the Device Model significantly simplifies the work involved in adapting to various hardware platforms. The memory range has also been extended, and supports up to 1MB (20Bit addressing), which means that developers of complex devices can implement larger applications. It is also possible with System B to define memory sub-segments, which enables the ETS to selectively download to the device (i.e. differential download). The most significant change from System 7 however is the increase of the size of the Group Object and Group Address Tables (the length field is 2 Bytes) shifting the theoretical limit from the previous 255 to 65535. The Association Table has also been extended and supports two formats: 2 Byte and 4 Byte addressing. The Association Table holds the indexes of the Group Address and Group Object Tables. Traditionally the Association Table allocated 2 Bytes per entry (one Byte for the connection number (TSAP) and one Byte for the object number (ASAP)) with a table length field of 1 Byte, imposing a limitation of 255 associations. With 4 Byte addressing and a table length of 2 Bytes, the Association Table can support up to 65535 associations (2 Bytes for TSAP and 2 Bytes for ASAP). The scaling of these tables implies certain challenges when adapting from a relatively small number (< 255) to upwards of many thousands when using a linear search algorithm (performance $O(N)$) in real-time (especially on an embedded platform with limited resources). One simple solution might be to simply increase the clock rate (if possible) however a more elegant and portable solution is to introduce a binary search algorithm and lookup tables. Using a binary search the average execution time is $O(\log_2(N) - 1)$ and the worst case is $O(\log_2(N))$ [4] where N is the number of addresses. As a logarithmic algorithm it implies that the number of N is no longer such a determining factor in terms of execution time when compared with a linear algorithm, as the size of the table grows.

	System 7	System B
Group addresses (max count)	254	65535
Group objects (max count)	255	65535
Loadable memory	~30 kB	~1MB

Table 1: Comparison of Resources for System 7 and System B

3 Solutions for Linux and WinCE

Platforms that support Linux or WinCE typically offer a greater range of options for application development than traditional embedded microcontroller environments, with limited Flash and RAM. These options can be seen in terms of available development languages, increased system resources and existing code bases in the form of both free (i.e. open source) and proprietary software solutions. Both platforms offer high-performance, are scalable, and support a wide range of hardware interfaces (in many cases this support is already “built-in” as part of a kernel driver or module). The KNX Stack for Linux (or WinCE) from Weinzierl Engineering contains the core of the KNX Stack for embedded controllers with a platform abstraction layer to enable support for various operating systems and their associated primitives and interfaces. The core stack code is implemented in C with both native Linux support (POSIX) and a cross-platform C++ implementation.

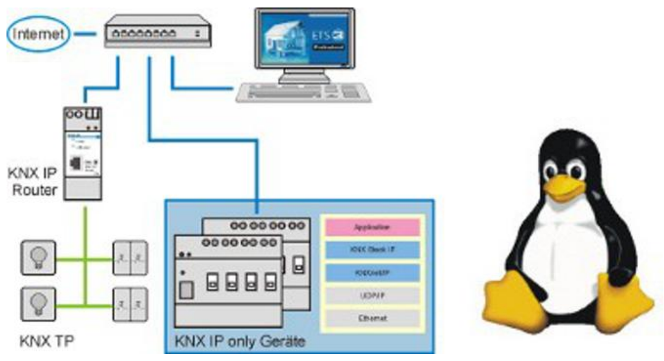


Figure 1: KNX Devices for Linux or WinCE implement the KNX device specification (in terms of Group Objects, Parameters and Management Procedures) and are managed via the ETS tool like any other KNX certified device.

3.1 KNX Stack Server

The KNX Stack has been designed according to the client-server model. A server in this context is defined as an application which owns the KNX resources (KNX Group Objects, ETS Parameters, Configuration Data etc) and shares them with one or more application clients. The server is responsible for interacting with the runtime KNX network, the ETS tool and application clients. A client does not have direct access to the KNX network, but requests a server's content or service function. Clients therefore initiate communication sessions with the server which awaits incoming requests [5]. Communication between the KNX stack server and application clients is realised via RPC (outlined in detail in Section 4.3 below). The client-server model has a number of advantages over a single application model. These include remote application clients (clients located on another machine), separate logging, monitoring or debug clients to ensure the integrity of both the client and server processes, and rapid prototyping and development, especially when coupled with an interpreted scripting language. Finally, strict determination of responsibility in terms of failure (crash) is immediate and obvious.

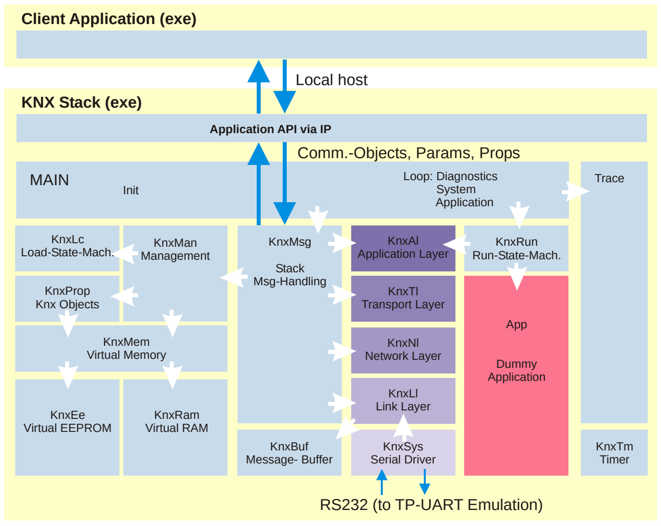


Figure 2: KNX Stack Client-Server Architecture (TP-EMU)

3.2 Connection to the KNX Network

The physical connection to the KNX Network currently has a large impact in terms of the device models available to complex KNX devices. For example, System B with its additional system resources is currently only specified in the KNX Specification for Twisted Pair, while there is only one device model available for IP (System 7: 5705). The coverage is outlined below in the Device Model / Media matrix.

	System 7	System B
Twisted Pair	0705	07B0
KNXnet/IP	5705	Not Specified

Table 2: Device Model / Media Matrix

Although it is technically feasible to increase the number of Group Objects above 255 for a 5705 (i.e. KNXnet/IP) client application by invoking multiple KNX stack servers each with its own individual address, the complexity at the configuration level (S-Mode, ETS) will probably preclude this as a viable option. Following is the list of currently supported KNX media access interfaces for Linux or WinCE from Weinzierl Engineering.

TP-UART Emulation

Twisted Pair UART Emulation (TP-UART Emu) is a hardware module with a transceiver and microcontroller that handles the real-time aspects of the bit stream to and from the transceiver and offers a non-real time UART interface to the host. The protocol is implemented according to the TP-UART chip protocol with several minor additions. The TP-UART Emu is responsible for the ACK decisions; all Group messages are accepted regardless of the Group Address and passed to the host; individually addressed frames are accepted according to a configured individual address. The TP-UART Emu module enables low-cost integration of the real-time nature of the KNX protocol with a non-real time process space (i.e. Linux userspace).

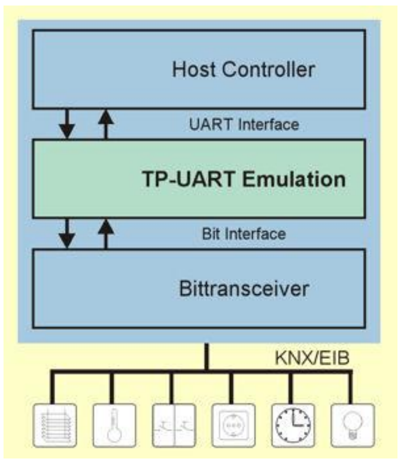


Figure 3: TP-UART Emulation Architecture

USB Interface

An alternative option is to incorporate a USB Interface component. This is the equivalent of embedding an USB Interface directly into your device. The KNX side is standard Twisted Pair and the device side is USB using the External Messaging Interface (EMI) protocol.

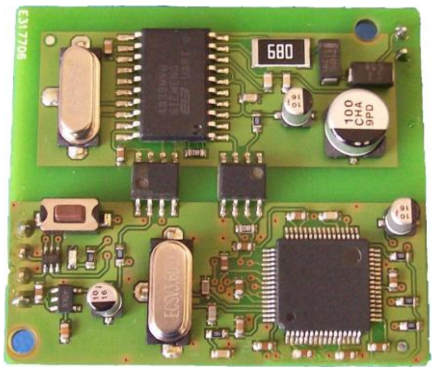


Figure 4: USB Interface Component

Internet Protocol KNXnet/IP

Devices that implement the KNXnet/IP protocol to access the KNX bus simply require (in addition to the KNX System Software) a standard Ethernet controller and TCP/IP protocol stack. These components are currently ubiquitous and can be found on (almost) every Linux or WinCE single board computer (SBC), and are readily found on much smaller platforms. IP-only devices are ideal for complex installations where Ethernet is available. System B is not currently defined for IP in the KNX specification. Work needs to be done in this area to enable the development of complex devices with resource requirements that extend beyond those of System 7.

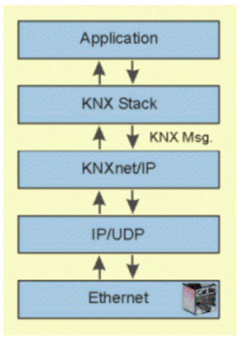


Figure 5: KNXnet/IP Stack Architecture

3.3 Remote Procedure Call

The KNX Stack Server implements a simple binary Request/Response protocol as a form of Remote Procedure Call (RPC). RPC is an inter-process communication mechanism that enables an application (the client) to invoke a function in another application (the server). The server can be resident either on the same machine (localhost) or on another machine reachable via a shared network (remote) [6]. The protocol uses standard TCP/IP socket communication which is reliable, well understood and very portable.

Communication with the KNX Stack Server is split into Stack API request/response pairs and asynchronous event indications. This is implemented in the server using two separate communication channels and listening threads, one for the RPC API (request/response from the client-side) and one for the Event Indications (from server). The communication architecture is shown below.

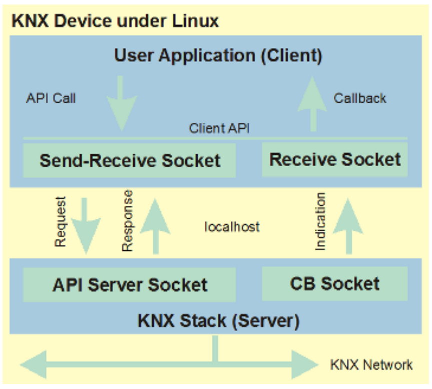


Figure 6: RPC-API Architecture

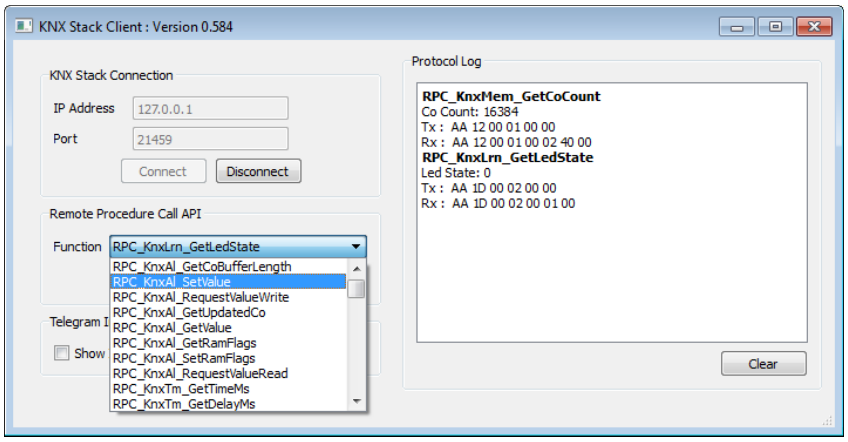
The user application (client) creates a TCP/IP socket connection with the KNX Stack (server). It can then use the RPC-API to call the KNX Stack API, as there is a one-to-one mapping. For example the KNX Stack API function

```
int KnxAl_GetUpdatedCo(unsigned int* pnCoNo);
```

is available in the RPC system as

```
int RPC_KnxAl_GetUpdatedCo(unsigned int* pnCoNo);
```

It is important to note that there is no information specified here in the RPC function call regarding the remote server connection. This information is saved by the RPC system on initialisation, which has the advantage of keeping the API signatures identical.
Our reference implementation is written in ANSI C and uses the Berkeley Sockets API (BSD Socket API). It is currently supported on both standard Windows and Linux environments. We also implement a client-side application to test the API, shown below.


KNX Stack Client : Version 0.584

KNX Stack Connection
IP Address: 127.0.0.1
Port: 21459

Remote Procedure Call API
Function: RPC_KnxLrn_GetLedState

```
RPC_KnxMem_GetCoCount
Co Count: 16384
Tx :  AA 12 00 01 00 00
Rx :  AA 12 00 01 00 02 40 00
RPC_KnxLrn_GetLedState
Led State: 0
Tx :  AA 1D 00 02 00 00
Rx :  AA 1D 00 02 00 01 00
```

Figure 7: Qt-based Cross-platform KNX Stack Client

3.4 Language Bindings

C++

C++ Language Bindings are available for the RPC API client software and directly wrap the C implementation. The advantage of developing with C++ is that the RPC error checking can be performed automatically and raised as an Exception. For example, to get the run state in C you would call the GetRunState(...) function and check for an RPC error:

```
run_state = RPC_KnxRun_GetRunState(app_no);

if (RPC_IsError())
{
      /* handle rpc error condition */
}
```

Whereas in the C++ bindings you can simply call:

```
Run::getState(app_no);
```

and handle the exception at some appropriate point in the call stack. This makes the C++ client software much more compact (visually) than the equivalent C version.

Python Language Bindings

Python is a popular interpreted scripting language [7]. The Python bindings for the RPC API are derived from the C++ implementation with boost::python. This leverages the auto-error detection and exception generation from the C++ software. Python lends itself well to rapid prototyping of the client side software.

4 KNX Product Database Generation

The KNX Manufacturer Tool (known hereafter as MT4) is used to create product data for ETS, the KNX engineering tool [8]. Both the MT4 and ETS4 software tools use the Extensible Markup Language (XML) as their primary source of data exchange. XML is a set of rules for encoding documents in machine-readable form [9] and its textual data format (ASCII, UTF-8 etc) is human-readable. The use of XML opens up new possibilities for working with and integrating with the ETS tools.

Developing a KNX device typically requires both the specification and implementation of hardware and software components. An equally important and no-less complex component is the development of the Manufacturer Product Database, which is the bridge into ETS and describes the Application Program along with its corresponding ETS Parameters, Communication Objects and Application Data.

KNX applications can specify downloadable binary data which is incorporated into the XML as a base-64 encoded ASCII string. The extraction of the binary data is complex, especially for the System B device model, and requires in-depth knowledge of the internal Memory Control Blocks in addition to knowledge of the Load Procedure Model and the use of Merge Points etc. This process would be overwhelming, if not impossible, without an automated tool to extract and generate the appropriate information. As part of our System B stack development package for KNX we have also implemented an accompanying component within our *Net'n Node* tool to auto-generate the required XML for an Application Program. The workflow is outlined in Figure X below.

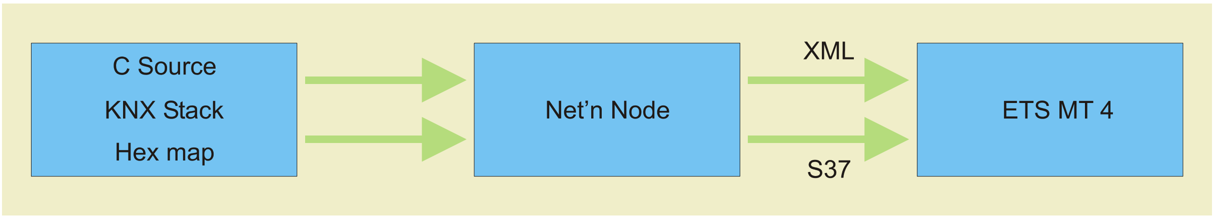


Figure 8: Manufacturer Product Database Workflow

Once the software development is complete information can be extracted (using the symbol table and application binary) to generate the corresponding Application Program XML. This information includes the Manufacturing Code, the Application Version, Relative Segment Size and Application Binary with Usage Mask. It also generates a skeleton for the Parameter Types and Communication Objects and defines the specific Load Control Procedure Merge Points required for System B. Once the XML has been generated the developer can create a MT4 project and add an existing XML file to the project (instead of creating a new application). From here the Parameters (types and references) and the Communication Objects (types, references and reference references) can be added as required. If the developer modifies the application binary after the Application Program XML has been generated it is possible to re-import the binary information as an S37 file. This information contains both the application binary and the corresponding Load Procedure Merge Points.

5 Conclusion

Complex KNX Devices have a diverse range of requirements that extend into all aspects of design and development, significantly influencing system level design choices. Using an open platform such as Linux with its wealth of available software can rapidly reduce the time to market for a complex device, and its inherent flexibility ensures a future-proof platform for additional products. Coupled with the new KNX Device Model System B these high-performance platforms can extend traditional applications to encompass complex application domains such as IP telephony and multimedia. An important factor when developing these devices is to ensure that their complexity is not reflected in the complexity of development. To this end, Weinzierl Engineering has a complete range of KNX System Software from embedded controllers to desktop servers with a toolchain and workflow to ensure complete integration into the KNX environment.

Bibliography

[1] KNX handbook version 2.0, KNX Association, Brussels, 2009

[2] For more information see http://www.weinzierl.de

[3] Kyselytsya, Yuriy; Weinzierl, Thomas: Implementation of the KNX Standard; Tagungsband KNX Scientific Conference November 2006

[4] Wikipedia: Binary Search Tree

[5] Wikipedia: Client-server model

[6] Wikipedia: Remote Procedure Call

[7] Python.org

[8] http://www.knx.org/

[9] Wikipedia: XML