Development kit for EIB/KNX devices based on the TP-UART chip

Key words: Bus access unit, EIB/KNX stack, TP-UART, ETS EIB Tool Software;

1 General information
In order to connect a device to EIB or KNX bus, an appropriate bus access unit has to be selected. Today, most devices use a standard BCU or BIM as the bus interface. Because of various technical restrictions and the high cost of components, this solution is not suitable for many applications, especially for those with a high number of units. An alternative is the TP-UART chip from Siemens (Twisted Pair Universal Asynchronous Receive and Transmit IC). This chip is an integrated circuit for bus access and provides the physical layer and a part of the link layer of the EIB/KNX stack for twisted pair. The rest of the link layer and the upper layers of the communication stack must be realised by the firmware of the microcontroller being used.

For this purpose we have designed a new operating system for EIB/KNX devices based on TP-UART. It includes, beneath the communication software, a complex memory management system to emulate the EEPROM and RAM areas which are directly addressed by the
EIB/KNX management tools but are not available on most microcontrollers. Figure 1 shows the typical device architecture.

Various device models (mask versions) are defined within EIB/KNX. They may be used to implement new system software and are supported by the ETS (EIB Tool Software) program. Currently, for twisted pair we have implemented the device models 0012 (BCU1), 0021 (BCU2) and 0701 (BIM M 112). Additional versions will be implemented as soon as they are approved standard for EIB/KNX.

We offer this firmware together with the PC program *Net’n Node* for development and test purposes. The package provides you with a complete environment for developing new low cost EIB/KNX devices. The most important features of the package are:

- Complete source code in C
- C-Level debugging of system and application modules
- Evaluation board available
- Support of device models 0012, 0021 and 0701
- Net’n Node test environment
- Low cost hardware design

The following paragraphs describe the architecture of our EIB stack implementation, giving an overview of the development process and the integration of new applications into the ETS database.

![Diagram](image-url)
2 EIB/KNX-Stack

The communication stack is implemented according the current EIB/KNX specifications. The structure of this stack is shown in Figure 2, and the individual modules are described in the following paragraphs.

![EIB/KNX stack diagram]

Figure 2: The EIB/KNX stack

2.1 Link Layer

The data link layer is implemented as a combination of the TP-UART chip and the “KnxLI” module of the firmware. Although the TP-UART chip handles some of the time critical parts of the protocol, especially those of the sending process, the firmware must also perform its tasks on time.

The sending part is quite simple since the microcontroller never sends a telegram directly to the bus. It sends the telegram together with certain control information to the TP-UART memory. Sending on the bus does not start until the entire telegram has been stored in the TP-UART chip and the bus is free.

In the receiving direction, the TP-UART does not store the telegram but sends each character to the microcontroller as soon as it is received. The controller not only has to accept these bytes but must also handle them. It has to decide whether or not an acknowledge must be sent before the telegram is received completely. The firmware handles these duties using various state machines. To determine the addressing mode, it checks the address table programmed in the non-volatile memory.

Compiled as model 0021 the device can act as polling master and slave (EIB fast polling) and is able to detect errors on the bus. All accesses of the link layer to the microcontroller
hardware registers are collected separately in a very small module called the “KnxSys”. Thus it is quite easy to use this firmware with different controller families. The API of the module only has the following four functions:

```c
void  KnxLl_Init(UCHAR nMode);
void  KnxLl_Close(void);
UCHAR KnxLl_Read(UCHAR* pBuffer);
BOOL  KnxLl_Write(UCHAR* pBuffer);
```

### 2.2 Network Layer

Since the network layer is already medium independent, this and all upper modules can be used in conjunction with a link layer based on a low functional transceiver. It can also be used e.g. for a powerline medium.

The task of the network layer of EIB/KNX devices is quite simple. In the receiving part, it checks the addressing type of the telegram and sets the corresponding service code. In the sending part, it performs the same task vice-versa. All service codes used for the stack are defined according to the EMI2 external message interface.

### 2.3 Transport Layer

The most important part of the transport layer is the handling of the connection-oriented communication which is used for management purpose. It is implemented according the EIB/KNX standard in different styles dependent on the used device model.

The group oriented part of the transport layer works connection-less and is implemented in a separate module called KnxTlg.

### 2.4 Application Layer

The application layer is the most complex layer because here a large variety of services, which are identified by their APCI code (APCI = Application Layer Protocol Control Information), has to be handled. For this reason we have divided it into several modules.

#### 2.4.1 Group Communication

The group communication is the most important part of the application layer during runtime. In fact all other parts of the application layer are only used for management and diagnostics. So this part gets the name KnxAl. The data points in the EIB/KNX system are called communication objects. A communication object represents for example the value of a sensor or the state of a switching actuator. So KnxAl handles all GroupValueRead and GroupValueWrite requests received from the transport layer. The object descriptors are read from the loaded virtual EEPROM memory space. The object values can be stored in this EEPROM memory as well, but mostly they are located in the virtual RAM area.

Access to the communication objects from the user application is available via the convenient function call. The most important call is shown below:

```c
BOOL KnxAl_SetValue(UCHAR nCoNo, UCHAR* pBuffer);
```
BOOL KnxAL_RequestValueWrite(UCHAR nCoNo);
BOOL KnxAl_GetValue(UCHAR nCoNo, UCHAR* pBuffer);

As in the EIB/KNX specification no presentation layer is provided, the data format (byte ordering, etc.) at the application layer has to be in accordance with the EIB/KNX interworking standards (EIS).

2.4.2 Broadcast Communication (Physical Address)
The broadcast communication is used to assign a unique physical address to an EIB/KNX device. Our EibStack supports the PhysicalAddressRead and PhysicalAddressWrite requests and handles the learning mode. In device model 0021 and 0701, it also supports the corresponding requests using the serial number of the device. In this device models, writing of the physical address can be disabled by means of the “service control” property in the device object.

2.4.3 Interface Objects (Properties)
In the device model 0021 and 0701, so-called interface objects have been introduced to access device data without using fixed memory locations. Interface objects are used for the configuration of the system but can also be used by the application. Each device supports the following four system objects:

- Device object
- Address table object
- Association table object
- Application object

These objects along with their properties are implemented in our firmware. Additional user objects may be defined in the virtual EEPROM memory space. Any virtual memory can be used for storing the property values. The firmware handles the PropertyDescriptionRead PropertyValueRead and PropertyValueWrite requests. Access protection features are included as well as properties with arrays.

2.4.4 Memory Access
Memory access via the bus is directed toward the virtual memory of the device. Support is provided for the MemoryRead and MemoryWrite requests. In the device models 0021 and 0701, memory access is protected by means of load state machines and access protection.

2.4.5 Access Protection
From the point of view of the ISO/OSI reference model, access protection is a task of the session layer. As in EIB/KNX, no session layer is specified and the access protection has been assigned to the application layer. In our stack, it is realised in a separate module. According to the device model, it handles four or sixteen access levels with three or fifteen keys. The lowest access level has no key. The current access level can be set via the Authorise request and the keys can be modified by sending a KeyWrite service to the device.
2.4.6 User Data

The specification of the EIB/KNX stack permits user data to be sent directly to the application process. Our firmware also supports this feature by means of a special mechanism. The application can receive and send user data by calling the following API functions:

```c
UCHAR KnxMsg_ReadUserMsg(UCHAR* pBuffer);
BOOL  KnxMsg_WriteUserMsg(UCHAR* pBuffer);
```

In addition, the user can receive the Connect and Disconnect frames from the transport layer and the PollingConfirms.

2.5 Message Flow

For the message flow we have implemented a special module called “KnxMsg”. It routes all received telegrams through the stack to the destination module of the application layer and vice-versa. Figure 3 depicts the message flow structure, showing an example for sending a communication object value with the confirm frame.

The up and down directions are handled separately with one buffer, or “workbench”, for each direction. Telegrams that run through the network, transport and application layer stay in place without a copy operation.

![Diagram](image.png)

Figure 3: Message flow for sending an object value
2.6 Virtual Memory
For configuration, the EIB/KNX device models use fixed memory locations that correspond to the microcontrollers from *Motorola*. If using another controller family, the memory map will usually be different. In this case, the address space cannot be used directly but has to be mapped onto the physically available memory.

2.6.1 Virtual EEPROM
Standard EIB/KNX devices use EEPROM memory which can be accessed byte-wise. In contrast, many microcontrollers use flash memory for non-volatile storage. Flash memory can only be written from logical “1” to logical “0”. For the other direction, a complete block must be erased.

To be able to use our stack on a flash-based controller, we have implemented an EEPROM emulation module. It uses a special RAM page to modify the data and writes the information back to flash at the end of the writing process.

2.6.2 Virtual RAM
The RAM space is also emulated to allow use of the standard EIB/KNX memory map as seen from the bus side. It is used for the values of the communication objects and the properties. In the device model 0012, low RAM is allocated at start-up. Working as device model 0021 or 0701, the system software allocates RAM according to the allocation records defined via load controls.

2.7 The User Application
The user application is the part of the software that is written by the application developer. As is already familiar from standard EIB devices, three methods have to be provided. The *App_Init* function is called once at start-up and *App-Main* is called cyclically for as long as the device is running and the application has not stopped. If a power fail interrupt is detected, the *App_Save* function will be called once. The calls of the different parts of the application are controlled by a run state machine according to the EIB/KNX specification.

Since the application can be programmed into the device memory during production, you will not have any constraints in memory size on the part of the device model. When using device model 0701 you can also use the ETS to program your application code into the device memory. Parameters of the application are handled in the same way as in standard BCU applications.

All hardware resources (timer, memory, etc.) of the controller not used by the stack may be used by the user application.

2.8 System
As the firmware is intended to run on small platforms, the system functions are quite lean. Beneath the functions for communication with the TP-UART chip, a timer and a heap memory are also available to the user.
2.9 Debug Support

During the development process you have access to the complete source code. Therefore, you may use any debug features the selected microcontroller offers. A great number of controllers have a selection of debug features on the chip that you can access, e.g. via a JTAG interface. You may even use a circuit emulator to debug applications. Thus, you can watch the memory or set breakpoints both in the application code being developed and in the code of the operating system. In addition, we have included in the firmware the debugging features explained below.

2.9.1 Tracing

The “Trace” module is used to write debug information to an output device. The simplest output device is a free on-chip UART, which you may use to send your debug information to your PC. If you do not have a free UART, you may add an external one or use a software-UART.

To save memory and time of the mikrocontroller our trace system sends not complete strings to the port. Instead we write only IDs which will be resolved by our tool TraceMon. The tracing function can be called from any module. Figure 4 shows the debug output of the firmware during a memory read operation for one byte at address 0x0116.

![TraceMon](image)

Figure 4: The trace output during a memory read

In the source code, macros are used to call the tracing function. These macros can be switched on and off separately for each module that you want to debug. For example, the macros

```c
TRACE_TEXT("Value = ");
TRACE_UINT8D(nValue)
```
write a string with an argument to the debug device. In release mode, they are not compiled and do nothing.

2.9.2 Stack surveillance
The term stack in this paragraph does not refer to the communication stack but to the stack memory of the controller. This module is not only used to detect a stack overflow, but also checks, in any program cycle, the number of unused bytes in the stack memory. If this number decreases, a debug message will be generated. Such a message can be seen in Figure 4: “Unused stack = 242 bytes”. This means that in the last program cycle more stack space was needed than ever before since starting the controller. Currently, 242 bytes of the reserved stack size have never been touched.

3 Hardware design
Hardware design is a task assumed by the application development. Evaluation boards are available for testing the firmware and for the first phase of development. Figure 5 shows a board with the TP-UART circuit at the upper left edge. This part of the board is isolated from the microcontroller by means of optocouplers. The microcontroller used here is a MSP430 from Texas Instruments. It is located in the big socket in the centre of the photo. The right part of the board is only used for debugging.

Figure 5: The evaluation board with TP-UART, MSP430 and debug components
4 Busmonitor Net’n Node

Net’n Node is a PC program that runs on Windows computers. It is made for development and test purposes in the laboratory. Figure 6 shows a screenshot of the program.

![Screenshot of Net’n Node](image)

Figure 6: A screenshot of Net’n Node

Net’n Node is the second major element of our development kit. The bus monitor is not limited to displaying the activity on the bus. It is able to interpret received telegrams and allows telegrams to be sent on any OSI/ISO layer. Furthermore, it offers the capability of debugging individual bus devices. It is possible to read and write physical addresses and to analyse the state of a single EIB/KNX device. The memory data of a device can be uploaded, downloaded or stored on disk.

Figure 7 shows the *hex view* of the device memory. Other views give an interpreted representation of the data. Figure 9, for example, shows the tables as part of the device memory. These views can be used to create and edit a device description file in s19 format, which later may be imported into the ETS program. Another view helps to define the load control records used to fix the steps for loading the device. To build a new device from scratch, a wizard is available that will guide you through the different entries.

Figure 8 shows a dialog for testing the link layer acknowledge in a specified address range. Figure 10 shows a dialog for scanning, reading and writing EIB/KNX interface objects.
The program allows several ports to be opened simultaneously. Beneath the connection to the EIB/KNX Net’n Node also features an RS-485 interface and a serial terminal is available. Of course Net’n Node supports the new USB interfaces for EIB/KNX.

As a special feature, the program has the ability to work as a “PEI spy”. Using a special hardware adapter, you can eavesdrop on communication between other computers or microcontrollers with a BCU. This is often a great help in debugging devices and applications that use a serial PEI protocol.
5 Integration in ETS database

For the purpose of integrating a new application based on our firmware, an s19 file can be created with Net’n Node. As described above, a wizard and multiple views will assist you with the integration of all configuration items, including property descriptions and load controls for the device models 0021 and 0701.

After the s19 file is imported into the ETS manufacturer tool, the device description has to be completed by setting certain strings and defining the parameters. The amount of effort that must be put into certification for the application is the same as for devices using a standard BCU or BIM.

6 Conclusion

As you can see, our EIB/KNX system software together with Net’n Node is a comprehensive development kit for building new low cost EIB/KNX devices. Cost savings can be achieved as early as the development phase, in which you have full access to the system software and no restrictions in making full use of the debug features. But the most significant advantage you will have is the highly effective device design: you will need only one microcontroller and a TP-UART chip to get an EIB/KNX device running.

Our system software of course is fully certified by EIB/KNX for different mikrocontroller like the MSP430 for Texas Instruments or the ATmega family from Atmel.

Today numerous manufacturers use our system software for new developments. In addition we offer complete device development including hardware design and support for certification.

7 Bibliography