Application Report
SWRA497–December 2015

Using MSP on KNX Systems

Damian Szmulewicz

ABSTRACT

This application report introduces KNX systems and describes the resources available for creating a KNX application on MSP microcontrollers (MCUs). An introduction to KNX is given to empower you with a basic understanding of the overall KNX system. You can find detailed information on the KNX specifications on the KNX association website (knx.org). This application report also provides a description of the software and hardware available for developing the KNX application with step-by-step examples that navigate the tools for project creation, application debugging, and system testing of MSP MCUs.

Contents

List of Figures

List of Tables

1 Introduction

Automation systems are increasingly popular and important in buildings of all sizes and complexities. Convenience, safety, and energy efficiency are key factors driving the need for intelligent monitoring and control of building products. From lightning and blind controls to complex HVAC and energy metering and management systems, residential and commercial buildings are being equipped with smarter automation solutions. This trend is driving manufacturers around the world to release thousands of products for this market every year.

The following three main components compose a typical building automation system:

- Sensors
- A communication channel
- Actuators

A set of sensors collect and process data from the environment (temperature, humidity, and so forth). Based on this sensory information, a message is sent through a communication channel to a different part of the network where an actuator takes action (turns the ACC ON or OFF and so forth).

Consider a safety system of smoke detectors, intrusion sensors, and an alarm. If smoke or an intruder is detected, the alarm receives a message and makes a loud sound. The alarm must correctly interpret the messages received from both smoke and intrusion detectors. But what would happen if the smoke detector is from manufacturer A using a communication protocol A, while the intrusion sensor is from manufacturer B using communication protocol B? How could the alarm correctly interpret both? What if hundreds of smoke detector manufacturers each use a proprietary communication protocol? A common language among all automation components is essential for their interoperability. This is where KNX becomes relevant.

2 KNX Overview

KNX is a worldwide communication standard for home and building automation. The KNX Association owns the KNX standard. In 1999, members from the European Installation Bus Association (EIBA), the European Home Systems Association (EHSA), and BatiBUS Club International (BCI) joined forces and founded the KNX Association. KNX is approved as an international standard (ISO/IEC 14543-3), a European Standard (CENELEC EN 50090 and CEN EN 13321-1) ,and Chinese Standard (GB/T 20965). KNX not only defines the communication language but also provides the set of tools and certification required to ensure seamless interoperability.

As outlined by the KNX association, a KNX system provides the following benefits:

- KNX is an international standard (future-proof).
- KNX ensures interoperability and interworking of products through product certification.
- KNX stands for high product quality by requiring manufacturer compliance with ISO 9001.
- KNX has a unique manufacturer: independent Engineering Tool Software (ETS).
- KNX can be used for all application areas in home and building control.
- KNX can be used in different kinds of buildings.
- KNX supports several communication media.
- KNX can be coupled to other systems using gateways.
- KNX is independent from hardware or software technology.

KNX is a well-established protocol used by an increasing number of developers for more than 25 years. More than 40,000 KNX installers exist worldwide. More than 360 manufacturing are producing KNX-certified devices. More than 100 thousand different KNX products are on the market from 7,000 different product families.

This document provides an overview of the KNX communication system but is not intended to replace the full KNX specifications that can be purchased from the KNX association.

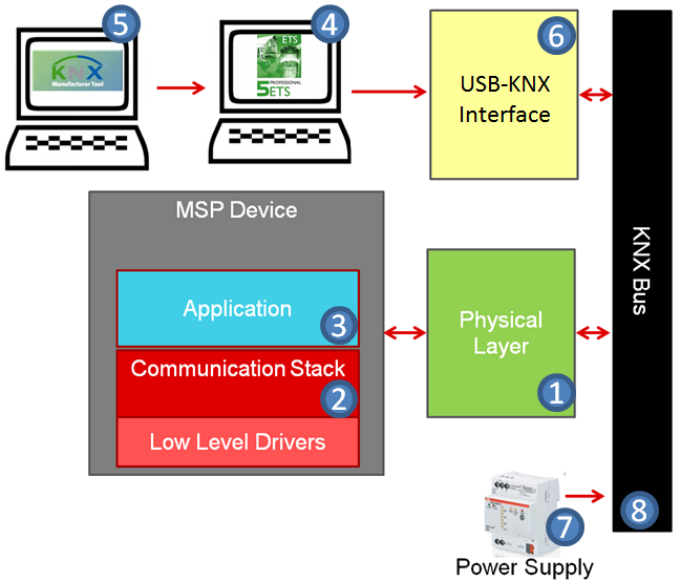
2.1 KNX Components

Figure 1 shows the eight main components that compose a simple KNX system:

1. The physical layer
2. The communication stack
3. The application program
4. The ETS product online configuration and installation
5. The Manufacturer Tool offline configuration
6. USB-KNX interface
7. KNX-certified power supply
8. KNX bus

The following sections discuss the details of each component.

Figure 1. KNX Components



2.1.1 The Physical Layer

To understand the need for a physical layer (PHY), the bus signals must be understood. As in Figure 2, KNX supports four types of media. This document describes the twisted-pair medium. For information on the specifications of all other media types, see www.knx.org.

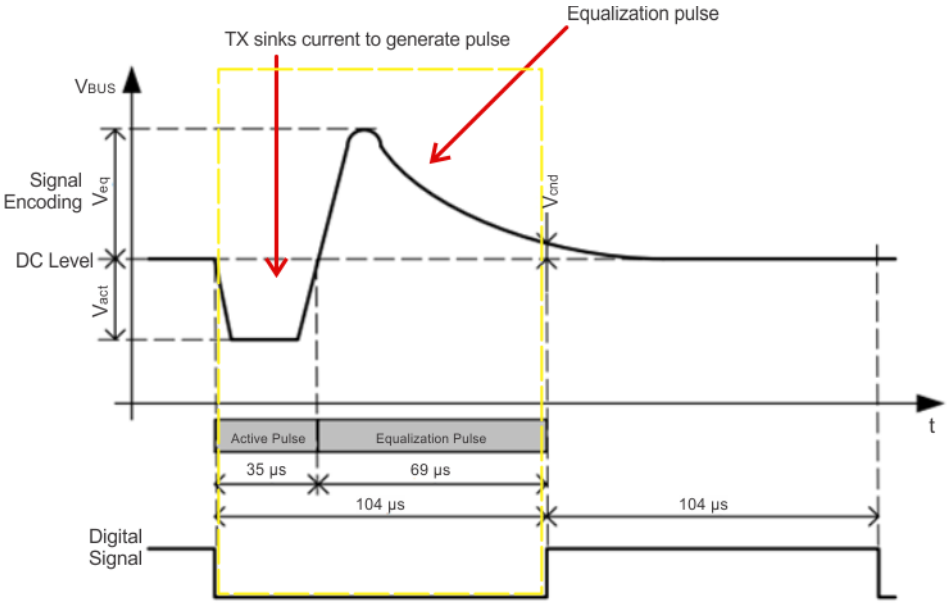
Figure 2. KNX Media

	Type	Information
Wired	Twisted Pair	UART 9600 baud ~90% of the market 1km max distance Collision detection and avoidance with priority
	Power line	1200 bits/s Collision control: random priority High level of background noise on the 230/400 V supply system, bus access cannot be related to the voltage level Collision problem has been resolved by the use of special time slots i.e. every mains coupler may only transmit during specified periods
	Wi-Fi	Encapsulated IP telegrams
Wireless	Ethernet	Encapsulated IP telegrams
	RF	868 MHz max power = 25mW 16.384 kbit/sec bitrate Single channel and multichannel

The twisted-pair medium is made up of a 24-V nominal differential voltage that operates from 20 to 30 V. A KNX signal on the bus can take one of two states: a logic 0 or a logic 1. A logic 1 is the DC level of the bus voltage, similar to a logic 1 in an I^2C bus (where the line is the same during idle state and a 1). A logic 0 is where the action of the KNX bus occurs. The encoding of the logic 0 occurs in two stages. The first stage is called an active pulse. An active pulse is a voltage drop of approximately 6 to 9 V in the bus voltage. This voltage dip lasts 35 µs and is generated by the transmitter sinking current from the differential lines.

Each active pulse is followed by an abrupt jump of the bus voltage greater than the DC level. This abrupt jump is followed by an exponential decay to the DC level with duration of 69 µs. This pulse is an equalization pulse. A choke indicator is typically built into the KNX power supply. This choke inductor generates the positive equalization pulse. The transmitter must ensure the correct levels and timings for a successful message transfer. Figure 3 shows the signal encoding on the KNX twisted-pair line.

Figure 3. KNX Signal Encoding for Twisted Pair



The receiver side requires a differential comparator with hysteresis. The receiver detects the beginning and end of an active pulse. The detection threshold for the start of the active pulse is −0.45 V (typical) less than the average bus voltage. The detection threshold for the end of the active pulse is −0.2 V (typical). Internal bandgap to suppress interference from the bus to the input comparator is also required.

Given the voltage level requirements of the KNX bus, analog components external to the MCU are required. The PHY is composed partially of the analog circuitry that enables the communication. KNX devices are bus powered. The PHY provides voltage regulators that output a stable 3.3-V DC for the application MCU.

KNX systems have an analog PHY that enables signal levels for communication. Some PHY options integrate a digital component that offloads digital functions from the application MCU, including collision control, repetition, acknowledgment, parity, and checksum. If the PHY does not include the logic for these digital functions, the application MCU must implement them.

To understand the importance of the digital functions of the PHY, consider the operation of collision handling. The transmitting device must constantly monitor the bus during transmission. Because a logic 1 is equal to the idle state of the line, only a logic 0 sent from another device can be detected. If two or more devices try to transmit at the same time, a collision occurs. The device sending 0 can continue while the device sending 1 must wait. This mechanism and strict time requirements ensure collision detection and handling.

Several PHY options are available in the market for analog-only and analog-plus-digital interfaces. The main trade-off between these interfaces is complexity and cost. The analog-plus-digital PHY provides the simplest solution on the application MCU but costs more than the analog-only solution. MSP supports both types of PHYs from certified vendors. The analog-only hardware interface is a *bit-based interface* and the analog-plus-digital hardware interface is a *TP-UART interface*. This document uses these terms.

2.1.2 The Communication Stack

Messages on the KNX bus are sent in packets called telegrams. As in Figure 4, strict timing requirements must be taken into account. Consider the case in which an event is triggered by the push of a button. A transmission may start after the bus has remained unoccupied for at least a time t1 = 5.2 ms. If this timing requirement is fulfilled, then the telegram is sent. When the transmission completes, the receiving devices must acknowledge the telegram within time t2 = 1.56 ms. If the transmitter does not receive the acknowledgment from the addressed devices, it retransmits.

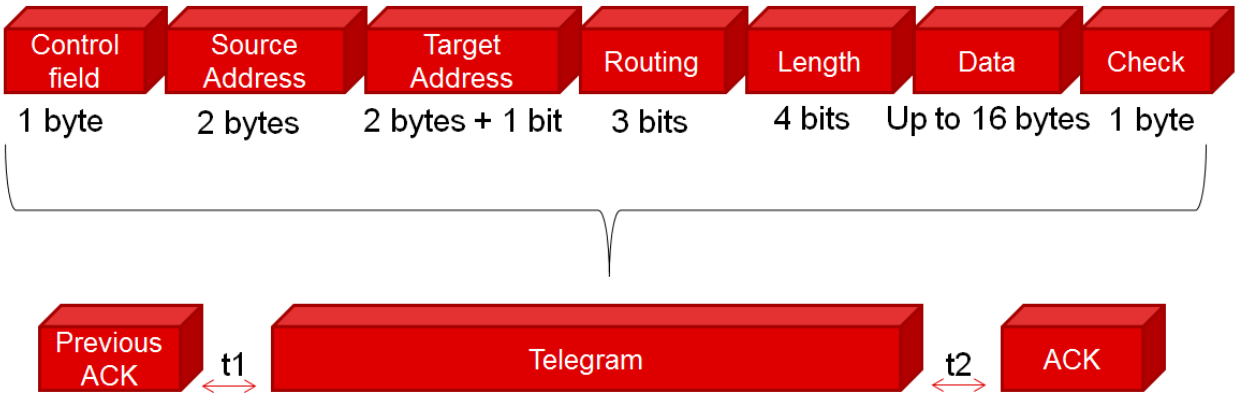
Figure 4. Timing Requirements Between Telegram Transmission and ACK

Information transmits as 8-bit characters with some overhead for data synchronization and error control as in Figure 5. A start bit indicates the beginning of a data byte. 8 bits of data, a parity bit, a stop bit and a 2-bit pause follow. 13 bits are required to transmit each byte of data.

Figure 5. Data Format on TP KNX Bus

Figure 6 shows the telegram is made up of seven fields. Sections 2.1.2.1 through 2.1.2.8 describe each field of the telegram and the coding of the acknowledgment.

Figure 6. Telegram Fields



2.1.2.1 The ACK:

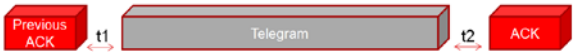


Figure 7 shows the acknowledgment is a byte of data that can represent three different responses:

- BUSY
- NACK
- ACK

Figure 7. KNX ACK Byte Description

D7	D6	D5	D4	D3	D2	D1	D0	Description
1	1	0	0	0	0	0	0	BUSY = device unable to process new information
0	0	0	0	1	1	0	0	NACK = Reception incorrect
1	1	0	0	1	1	0	0	ACK = Reception correct

2.1.2.2 The Control Field:

The control field is a byte that serves different purposes. If one of the addressed bus devices returns a NACK and the transmission repeats, a repeat bit in the control byte is set to 0 to indicate this is a retransmission. The repeat bit ensures that the receiving device executes a command only once. The control byte also sets the priority of the message. Figure 8 shows the functional description of the control field.

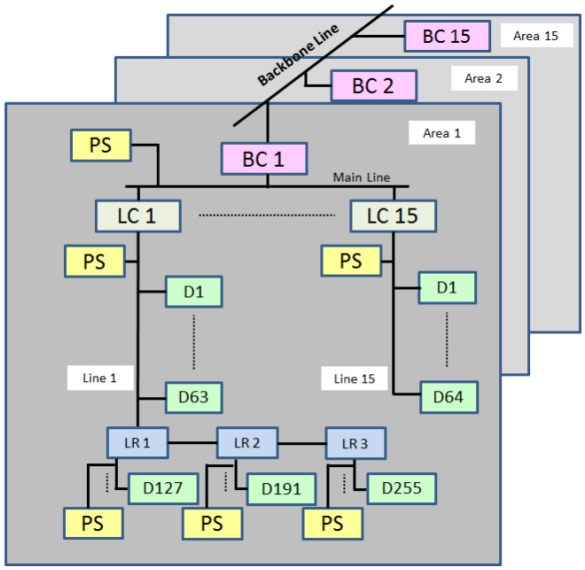
Figure 8. Bit Encoding for the Control Byte

D7	D6	D5	D4	D3	D2	D1	D0	Description
1	0	R	1	P	P	0	0	PP = TX priority, R = repeate, others = fixed
				0	0			System functions (highest priority)
				1	0			Alarm functions (alarm)
				0	1			High operational priority (normal)
				1	1			Low operational priority (auto)
		0						Repeat

2.1.2.3 The Source Address:

Each device in the KNX network has a 2-byte address. To understand KNX addressing, consider the KNX topology. Figure 9 shows an overview of a KNX network. Starting at the device level, up to 256 KNX devices (Dn) can be connected per line. Add line repeaters (LRn) to each line to increase the cable length. If more devices are used in the system, up to 15 lines can be connected together using line couplers through the main line, enabling 3840 devices. A maximum of 15 lines form an area. A backbone line expands the KNX twisted-pair bus. A maximum of 15 areas can be interconnected using backbone couplers (BCn) yielding a total of 57600 maximum total devices per KNX network. Each line and line segment must have its own power supply. The power supply specifications limit the actual maximum number of devices in the system.

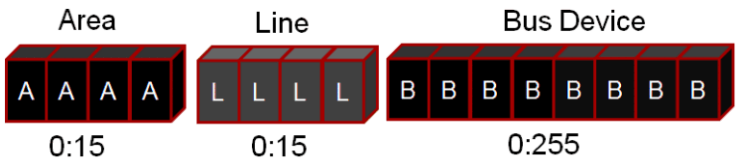
Figure 9. KNX Topology: BCn = Backbone Coupler, LCn = Line Coupler, LRn = Line Repeater, PS = Power Supply, Dn = KNX Device or Node



For more information on KNX topology, see *KNX TP1 Topology* (http://www.knx.org/fileadmin/template/documents/downloads_support_menu/KNX_tutor_seminar_page/basic_documentation/Topology_E1212c.pdf).

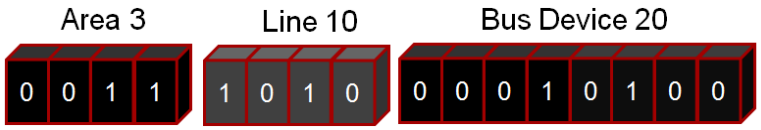
To identify a specific device on the network, the area, line and device number must be present in the source address field. Figure 9 shows if only the device address of D1 is given, the transmitter could be D1 on line 1 or D1 on line 15. Similarly, the area number must be specified. The source address contains the complete information of the transmitting device. As shown in Figure 10, the lower 8 bits are the individual bus device address. The low nibble of the most significant byte (MSByte) of the address specifies the line number and the most significant nibble has the area number.

Figure 10. KNX Source Address Contains the Area, Line, and Bus Device Number



Consider a device with individual address 20_{10} on line 10_{10} and area 3. Then the source address is given by 3/10/20 as shown in Figure 11.

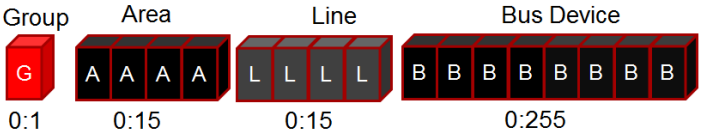
Figure 11. Source Address Example



2.1.2.4 The Target Address:

The target address contains the address of the device that is meant to process the transmitted message. Figure 12 shows that this address also contains the area, line, and individual address of a device on the KNX network but has an extra bit that specifies whether the address is intended for a single node or multiple devices. The extra bit is a useful feature of the KNX system. Consider a system where 1 switch is used to control 20 light bulbs in a room. Each lightbulb has an individual address, requiring the switch to send the same message 20 times addressing each target device. By setting the group bit to 1, the individual address is ignored and devices in the specified line of a specified area process the message. Twenty light bulbs can be controlled by a single message.

Figure 12. KNX Target Address



The target address is an individual address for point-to-point connections (connection-oriented communication). For multicast or broadcast addressing (connectionless communication), addresses called group addresses are used as target (receiver) addresses and have the structure shown in Figure 13.

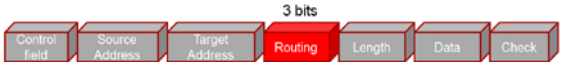
Figure 13. Group Address Coded on 15 Bits (Bit D15 is a Reserved Bit)

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
	Main Group				Subgroup										
	Main Group				Middle Group			Sub Group							

D7
T

The first row shows how the group address is represented in a 2-level hierarchy (main/sub), while the second row shows a 3-level hierarchy (main/middle/sub). There is no physical difference between a 2-level and a 3-level group address on the bus.

2.1.2.5 The Routing Bits:



The routing bits form a 3-bit counter to limit how far a message can be transmitted in the KNX network. The routing value is initialized by the transmitter and modified by the couplers in the system. Each coupler decrements the routing counter and passes the telegram as long as the value is greater than zero. If the counter is equal to zero, the message is not transmitted by the coupler. This feature suppresses short circuits between different lines, preventing message looping forever on the bus.

The filter tables in the routers (line couplers) do this filtering. To disabled the filtering, make the routing counter equal 7. If the routing number is 7, the couplers continue to transmit without decrementing the routing bits.

2.1.2.6 The Length Field:

The Length field has the number of useful data in the Data field. The length of data depends on the transported size of data and can vary between 1 bit and 16 bytes. The receiver uses the Length field to check correct data reception. Only data bits and not data flow or control bits (such as parity, start and stop, and so forth) are considered when calculating the Length field.

2.1.2.7 The Data Field:

The Data field contains the actual data of the telegram, such as *turn light on*. Messages are encoded into standardized data point types to ensure interoperability of devices. Data point types ensure that all devices communicate in the same language. Take a switch that controls a set of lightbulbs. The bulb has only two states: On and Off. In the list of KNX data point types, there is a type called DPT_Switch. Figure 14 shows the description of DPT_Switch as well as other 1-bit data types. DPT_Switch is a 1-bit message that represents either an On or Off state. This type can be used for the light switch.

Figure 14. KNX Datapoint Type Example

Format:	1 bit: B_1
octet nr	1
field names	b
encoding	B
Range:	b = {0, 1}
Unit:	None.
Resol.:	(not applicable)

Datapoint Types

ID:	Name:	Encoding: b		
1.001	DPT_Switch	0	=	Off
		1	=	On
1.002	DPT_Bool	0	=	False
		1	=	True
1.003	DPT_Enable	0	=	Disable
		1	=	Enable
1.004	DPT_Ramp	0	=	No ramp
		1	=	Ramp
1.005	DPT_Alarm	0	=	No alarm
		1	=	Alarm
1.006	DPT_BinaryValue	0	=	Low
		1	=	High
1.007	DPT_Step	0	=	Decrease
		1	=	Increase
1.008	DPT_UpDown	0	=	Up
		1	=	Down
1.009	DPT_OpenClose	0	=	Open
		1	=	Close

Applications require varying data point types. For a list of data point types, see *Interworking* (http://www.knx.org/fileadmin/template/documents/downloads_support_menu/KNX_tutor_seminar_page/Advanced_documentation/05_Interworking_E1209.pdf)

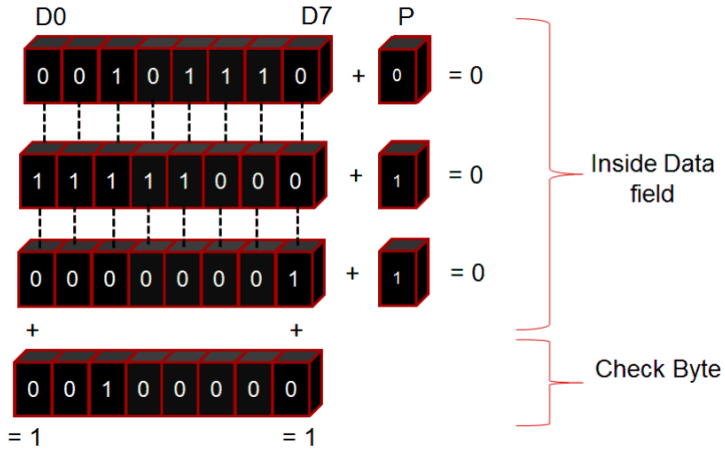
2.1.2.8 The Check Field:

The KNX communication protocol offers two levels of error checking. The first is at a byte level. As shown in Figure 15, each byte of data in the telegram has its own even parity bit. That is, the parity bit P gets the value 0 or 1 to make the binary sum of all bits (D0:D7 and P) equal to 0.

Figure 15. Even Parity for Each Byte of the KNX Telegram

The last byte of the telegram (called the check field) does the second check. This field is created by checking all characters of the telegram for odd parity for each bit position. Consider having 3 bytes of data as shown in Figure 16. Each byte has its own even parity bit inside the data field. Each bit of the check byte is calculated based on the sum of each bit of all the data bytes for that bit position so that the binary sum equals 1.

Figure 16. Byte and Telegram Level Error Check



2.1.3 The Application

Along with the KNX communication stack, the application software runs and enables the specific function of the product. Based on the selected PHY, different device resources may not be available. For example, the analog-plus-digital PHY uses a UART interface that is no longer available for the application. An initial assessment of the resources required for the application is essential for the selection of the best device and stack option.

2.1.4 The ETS Tool

The ETS is a manufacturer-independent configuration software tool. ETS is the only configuration tool that can be used in the installation of home and building automation in KNX systems. According to the KNX organization, ETS presents the following advantages (see http://www.knx.org/in/software/ets/about/index.php?navid=948232948232):

- Maximum compatibility of ETS software with the KNX standard is ensured.
- Product databases with certified products from all KNX manufacturers can be imported in ETS .
- Backward compatibility of ETS to product data and projects of earlier ETS versions (as far back as ETS2) safeguards your working results and allows editing.
- All designers and installers use the same ETS tool for every KNX project and with every KNX-certified device. (Reliable data exchange is ensured.)

ETS contains a database of certified KNX products to create an automation system in a home or a building. With the ability to assign spatial characteristics and device addresses, ETS simplifies the system installation while maintaining cross-manufacturer compatibility. ETS directly configures devices through the KNX bus; special hardware that connects a computer to the bus is required. Section 2.1.6 describes this hardware. To configure a new KNX product in ETS, the manufacturer tool must create a database entry.

For more information, tutorials, and to download a demonstration version of ETS, see the ETS learning campus at http://wbt5.knx.org.

2.1.5 The Manufacturer Tool

KNX products must be certified and available as a database entry for installation. The manufacturer tool enables developers to create such an entry for their products. The manufacturer tool is required to create and test ETS product entries and also to have products certified by the KNX Association. When the product is certified, manufacturers provide the entry as downloadable product catalogs for installers to use.

2.1.6 The USB-to-KNX Interface

To configure devices on the KNX bus using ETS, the computer must be able to communicate over the twisted-pair bus. For this purpose, the device in Figure 17 named UIM-KNX 42 from Tapko is available. UIM-KNX 42 establishes a bidirectional data connection between the PC and the KNX bus. The device enables addressing, setting parameters, visualization, protocolling, and diagnosis of bus devices. The USB connector is galvanically isolated from the KNX bus. With this KNX–USB interface, every device in the bus can be addressed. The communication between the KNX–USB interface and the connected devices is handled through flexible common EMI protocol through the UIM-KNX 42 device. This protocol is designed for current and future applications.

Figure 17. USB-to-KNX Interface for PC-to-Device Communication

2.1.7 The Certified Power Supply

As in Figure 9, each line in the KNX topology must have a certified power supply. The power supply provides power to the network and is also used for communication. Figure 18 shows an example of a certified power supply. The power supply is typically rated at 30 V and contains a choke inductor that is responsible for the equalization pulse in Figure 3. Power supplies with different current ratings are widely available in the market. For help choosing a power supply, contact TI experts on E2E™ (http://e2e.ti.com/support/microcontrollers/msp430/).

Figure 18. KNX Power Supply IPS640

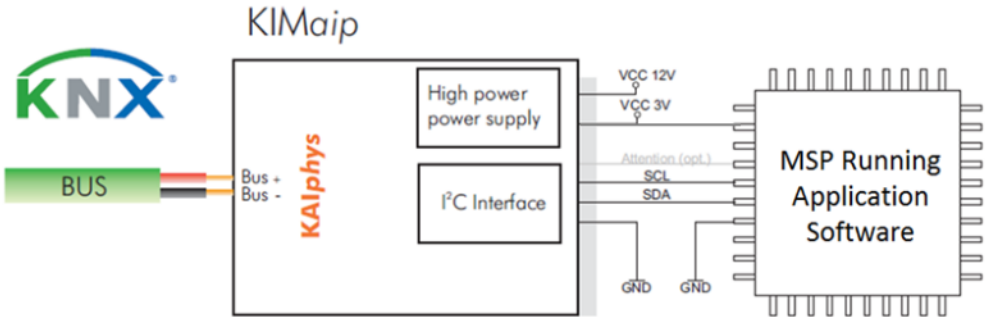
3 The MSP-Tapko Offering

TI has partnered with Tapko technologies to enable customers to develop building automation solutions with KNX. This section describes the hardware and software available to start developing KNX-enabled MSP home and building automation applications. Hardware and software solutions are available for varying needs. The number of expected annual units is a key factor in determining the most cost-effective solution. This document provides an overview of the options. For further KNX support, request help at the MSP Low-Power MCU Forum (http://e2e.ti.com/support/microcontrollers/msp430/).

3.1 Option 1: KIMaip

The fastest way to market a product is to use KIMaip interface from Tapko combined with a low-power MSP microcontroller running the application. KIMaip is an easy-to-use interface module for connecting to the KNX bus. Access to the KIMaip module from the application controller is achieved through the I^2C bus. KIMaip consists of a microcontroller running the certified KNX communication stack (KAIstack) and the high-performance KNX interface to the KNX bus (KAIphys). This design enables direct use of the high-power power supply from the bus. This module is intended to directly connect to an application controller and is optimized as basis for devices. This module is suited for applications that do not require galvanic isolation. The KIMaip module reduces time to market by making it easy to implement KNX functionality and removing overhead in the application development. Figure 19 shows the system diagram when using KIMaip.

Figure 19. KIMaip Option



The KIMaip interface has the following characteristics:

- Offers KIMaip bus module is the slave and the MSP controller is the master.
- Represents pinary data of messages on I^2C.
- Offers access to group communication and interface objects.
- Offers access to ETS-configurable parameter area.
- Object data is stored in external user application controller.
- Offers optional *attention KNX data received* pin.

Some of the KIMaip features are as follows:

- Offers high-efficiency KNX physical layer KAIphys provides maximum 35 mA at 12 V or 30 mA at 3-V output power for application MCU.
- Offers direct connection of application controller and application hardware to module power supply.
- Offers configuration through free downloadable generic ETS database entry.
- Offers 253 group objects.

KIMaip supports all KNX data types and provides immediate indication when data is received with object data. The application controller does not execute any KNX-specific code. The KIMaip generates a telegram immediately after reception over I^2C. KIMaip is neither a bit-based (analog only) nor a TP-UART (analog-plus-digital) interface.

3.2 Option 2: KAIphys and KAIstack With KAIlink-BIT

KAIphys is a bit-based interface solution. KAIphys is the first KNX interface based on standard components, available for all users, and implemented without an ASIC. KAIphys is a superior solution that enables high flexibility in usage combined with a high level of signal processing quality and a high energy from the bus. This product allows cost effective KNX solutions. The fact that no custom-specific ASIC is required and the possibility of choosing among various options in various modules offer an extraordinary potential for streamlining the hardware design of future products. KAIphys offers maximum performance with high flexibility. The innovative circuit is the hardware component of the technology platform KNX Advanced Interface (KAI) for KNX-enabled bus devices. KAIphys and KAIstack form the basis for complete KNX devices. Hardware and software components of KAI can be adapted to conditions. KAIphys is KNX certified. For implementing a solution with KAIphys, licensing of the module is available from Tapko.

KAIstack is the main software component of KAI and provides the complete functionality required for KNX devices. In other words, KAIstack contains elements required by the KNX standard and is certified in several different configurations implemented in close dependence on the ISO/OSI reference model. The clear structure separates between application-relevant parts, modular-communication-stack-internal parts, media-dependent parts, and target CPU-related issues. Advanced implementation methods allow highly-efficient coding that leads to optimized resource usage. An application development with KAIstack relieves the burden of early decisions in the design process by simple configuration of the stack while increasing the reliability of the system and improving the stability of the device.

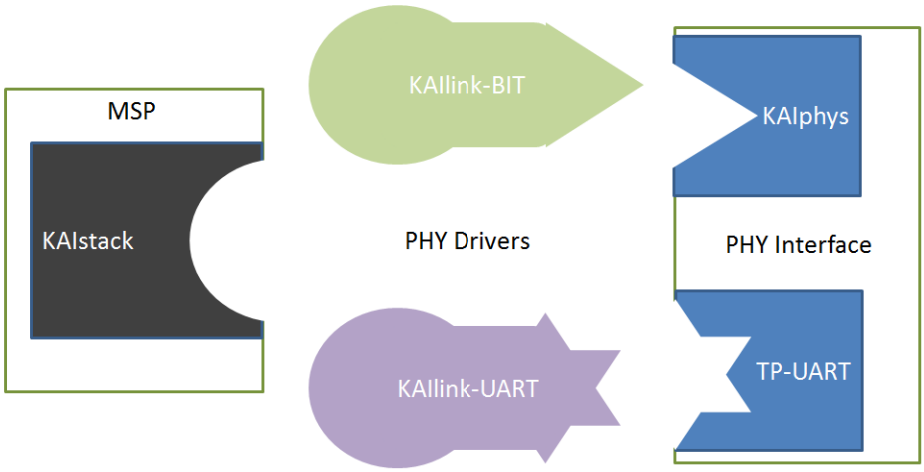
KAIlink-BIT is the software driver that links the physical bit-based interface (KAYphys) to the software stack (KAIstack). Combining KAIstack with KAIlink-BIT gives a full software solution.

3.3 Option 3: TP-UART and KAIstack With KAIlink-UART

Several hardware TP-UART interface chip solutions, such as SIEMENS TPUART2, ON semi NCN5120, and ELMOS E981.03 are available in the market. PHY media modules are available for each chip solution. Tapko offers certified plug-and-play media modules that enable fast prototyping and development.

As in Section 3.2, KAIstack is the KNX communication stack. From the software perspective, the difference between option 2 and the TP-UART interface option is the driver for interfacing with the physical layer. KAIlink-UART combined with KAIstack offers the complete software packet for developing KNX on TP-UART. To better understand the role of KAIstack and KAIlink, Figure 20 shows how these software packages fit in the KNX solution. For the KAIstack to interface with the bit-based PHY, KAIlink-BIT is required. For the KAIstack to interface with the TP-UART PHY, the KAIlink-UART driver is required.

Figure 20. KNX Software and Hardware Ecosystem



3.4 Option 4: Customer Stack Software and Hardware

For high-volume applications where an available module for the physical layer is not an option, TI and Tapko can provide a fully customized KNX solution. This option lets you optimize everything for a specific application but presents the highest cost. For more information about this option, contact the MSP experts at the E2E forum (http://e2e.ti.com/support/microcontrollers/msp430/).

4 Getting Started With KNX on MSP

This section describes the hardware and the software to start developing KNX-enabled application and shows how to create a new KNX application using the available tools.

4.1 Hardware

Figure 21 shows the minimum hardware required for KNX development.

Figure 21. Minimum Hardware for KNX Development With MSP

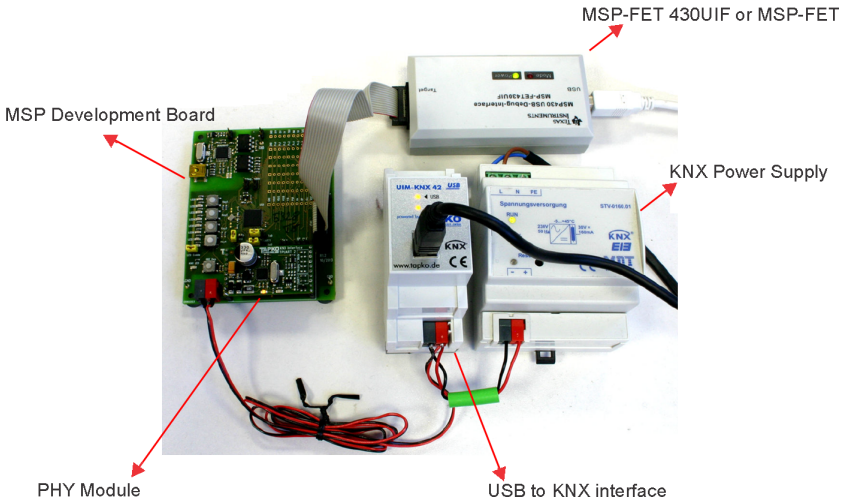


Table 1 lists where you can obtain more information for each of the components in Figure 21.

Table 1. Hardware Component Resources

Component	Link
MSP-FET	http://www.ti.com/tool/msp-fet
USB-to-KNX interface	http://www.tapko.de/en/uim-knx42
MSP development board and PHY module	http://www.tapko.de/en/kaistack
KNX power supply	http://www.tapko.de/en/ips640

Additional software and hardware resources may be available as TI designs or application reports. Search for KNX at www.ti.com for latest resources.

4.2 Software

To start evaluation, Tapko provides a free version of KAIstack and KAIlink with the following limitations:

- Offers only sixteen group addresses, sixteen associations, and sixteen communication objects
- Offers no interface objects
- Sets network layer Rout-Count to 1
- Offers no transport layer repetitions
- Prevents ETS from changing the physical address
- Supports only one device derivative

Download the software installer from http://www.tapko.de/en/kaistack-for-ti.

In addition to the KAI demonstration, a valid license of IAR Embedded Workbench for MSP is required. For a time-limited evaluation license of IAR, navigate to http://supp.iar.com/Download/SW/?item=EW430-EVAL.

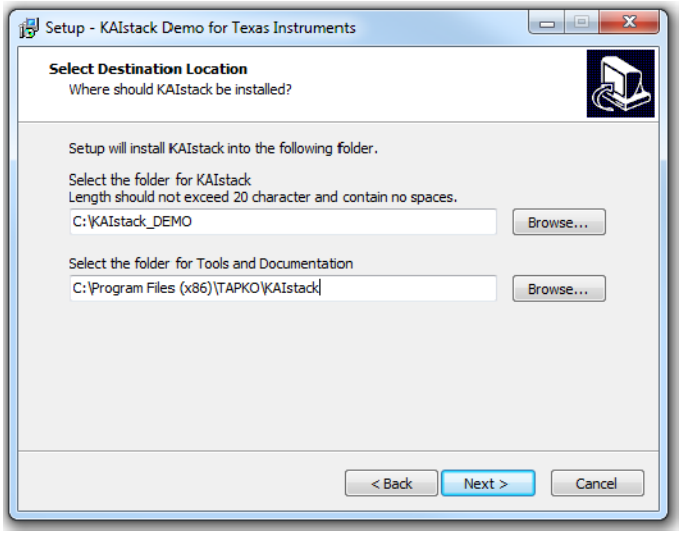
NOTE: Due to code size, the full version of IAR Embedded Workbench for MSP is required.

4.2.1 Demonstration Software Overview

This section gives an overview of the evaluation software package from Tapko. The KAIstack installer creates two directories. Figure 22 shows the install screen where you can select the destination folder for each generated folder.

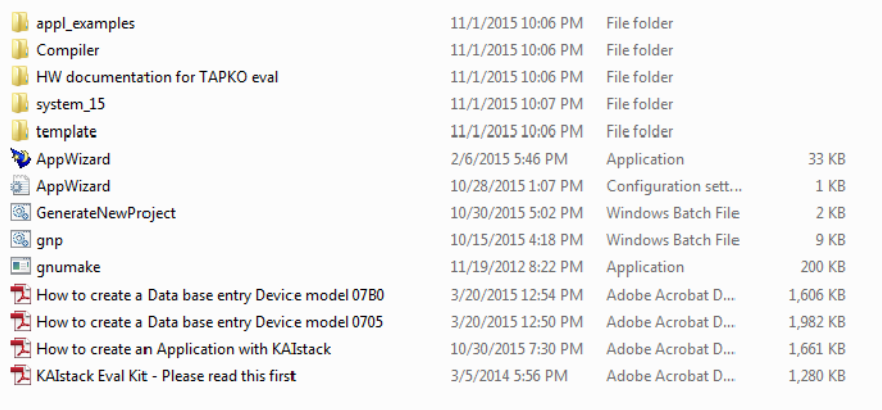
NOTE: The directory path for KAIstack cannot have any blank spaces. If there is a space on the path of KAIstack, the installer completes successfully, but the compilation of the project fails later. TI recommends installing KAIstack in the root of a drive (such as C:\KAIstack_Demo).

Figure 22. Installation Directories for KAIstack, Tools, and Documentation



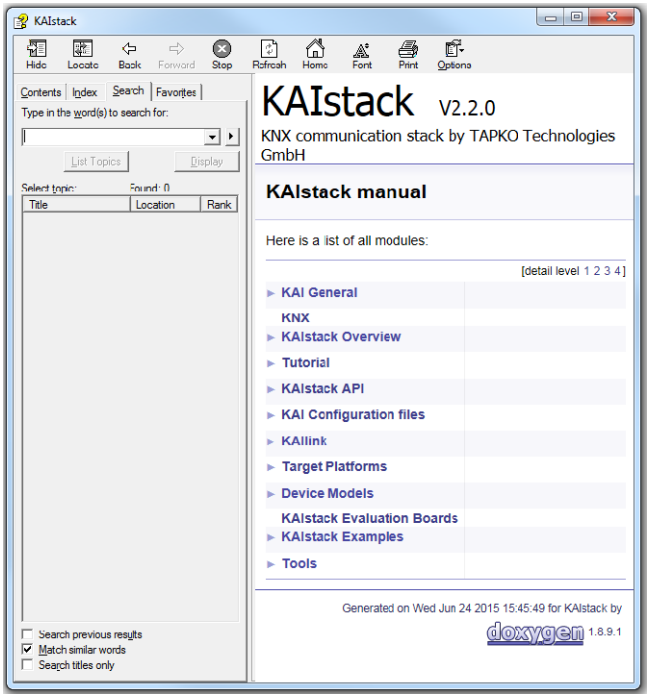
The first folder is the KAIstack directory that contains five directories, an AppWizard to help create a new project, a GNU make file and documentation to guide users in the creation of ETS entries and other KNX-related activities. See Figure 23.

Figure 23. KAIstack Demonstration Directory Structure



The AppWizard is an application that creates the framework for starting a new KNX project. Section 4.3.1 provides the step-by-step instructions for using the AppWizard. The system_15 directory contains application-independent files required for the stack to run correctly. TI does not recommend modifying these files. The Compiler directory contains the files required by IAR to compile and link the KNX project. TI does not recommend modifying any of these files. appl_examples provides sample code that lets you generate functional KNX projects; this code is a good reference when developing a new project. The second directory of the installer is for tools and documentation. Figure 24 shows the KAIstack reference manual.

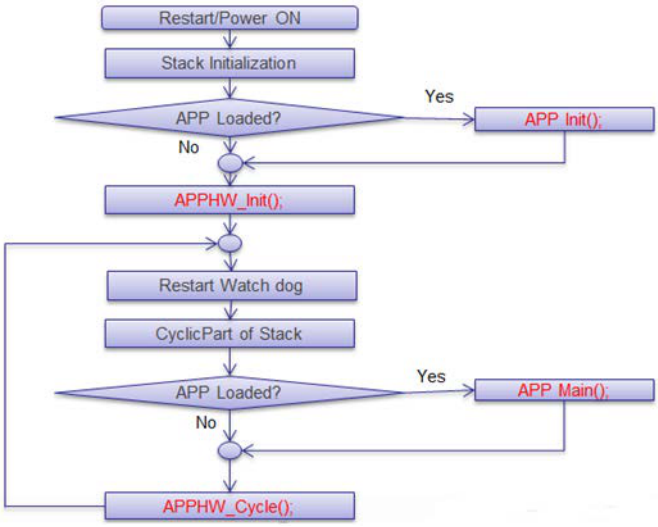
Figure 24. KAIstack Reference Manual



4.2.2 Stack Software Flow

This section gives an overview of the software flow provided in Figure 25. For a complete description of the stack, review the KAIstack API in the KAIstack reference manual.

Figure 25. KAIstack Software Flow



When a device running KAIstack is powered up, the stack is initialized. This initialization includes all low-level drivers including KAIlink. If an application is loaded into the device, the KAIstack scheduler calls the APP_Init() function. The user code in APP_init() initializes the application variables. In KNX, the device model defines memory areas that are accessible through the bus. These memory areas are mapped to the physical memory range using pointers. This mapped area is the virtual BCU_RAM memory. The virtual BCU_RAM memory (RAMflags and communication object values) is cleared before APP_Init() is called.

The KAIstack scheduler calls APPHW_Init(). APPHW_Init() is similar to APP_Init() but it initializes the application peripherals instead of the application variables. You can the set up timers, serial interfaces, and analog frontend in APPHW_Init(). After initialization completes, the software scheduler calls APP_Main() cyclically. By default, the stack calls APP_Main() as fast as possible (in the order of hundreds of microsecods) and depends on bus activity. You can make this delay between calls slower, but it must not be more than 10 ms to ensure that the application reacts to all messages without any blocked communication objects. Similarly to APP_Init() and APPHW_Init(), the cyclic part of the program has APPHW_Cycle() and APP_main(). APPHW_Cycle() is always called but APP_Main() is called only if the application is loaded and configured into the device.

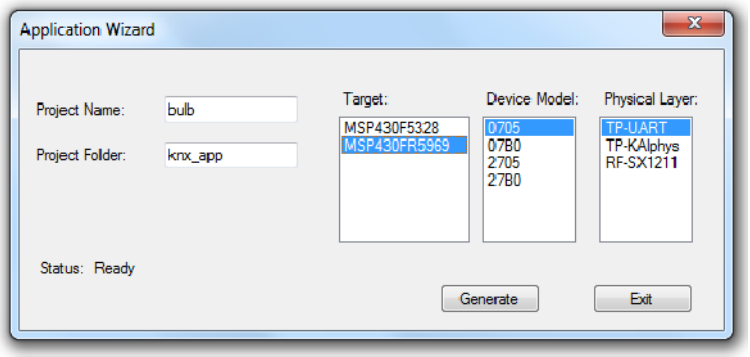
4.3 Creating a Custom KNX Application

After installing the KAIstack demonstration, use AppWizard to create a new KNX application. This section describes the steps required to create a new KNX project and to add an application to it. TI recommends installing IAR (http://supp.iar.com/Download/SW/?item=EW430-EVAL) and ETS (http://wbt5.knx.org) before continuing.

4.3.1 AppWizard

The following steps show how to start creating a KNX project on MSP. Figure 26 shows the interface of AppWizard.

Figure 26. AppWizard Graphical User Interface



1. Select the project name and project directory.
2. Select the target device.
3. Select device model 0705 or 07B0 (bothe aregreater than 250 data points)

NOTE: Device models that start with a 2 are for the RF interface and beyond the scope of this document. For more information on device models, see device models in the KAIstack reference manual.

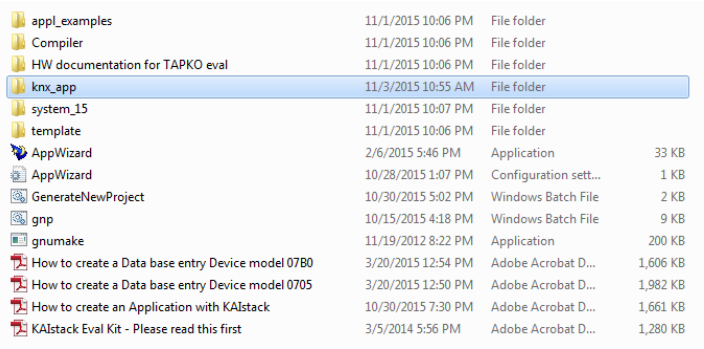
4. Select the physical layer: TP-UART or TP-KAIphys (bit-based PHY).

NOTE: RF-SX1211 is for the RF interface.

5. Click *Generate*.

NOTE: A directory with the folder name is created inside the KAIstack folder as in Figure 27. This directory contains the required files for compiling and debugging the KNX project.

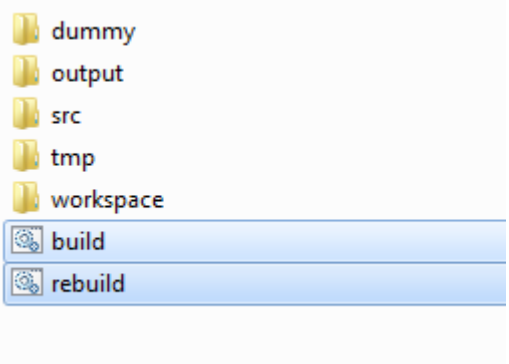
Figure 27. AppWizard Generated knx_app Directory



6. As in Figure 28, the following directories are created inside the new project folder:
   - dummy
   - output
   - src
   - tmp
   - workspace

   Two Windows® command scripts (.cmd) are creating for building your project.

Figure 28. Content of knx_app Generated by AppWizard



4.3.2 Understanding the Generated Files

This section describes the purpose of each file. Section 4.3.2.1 describes how to modify these files to create a new application. The dummy, output ,and tmp directories are initially empty. These folders are used by the build and rebuild command files when the project is compiled. The most important file created by build is the debug file (.d43) that is inside the \output directory. Section 4.5 describes the use of the debug file. The \scr directory contains the source and header files required to run the application. Figure 29 shows the six files created inside \src. You can add more files to \src for the application.

Figure 29. Files in the src Folder

4.3.2.1 app.h

This file contains the definitions of parameters required for the KNX to compile correctly. First a set of identification numbers is defined as follows:

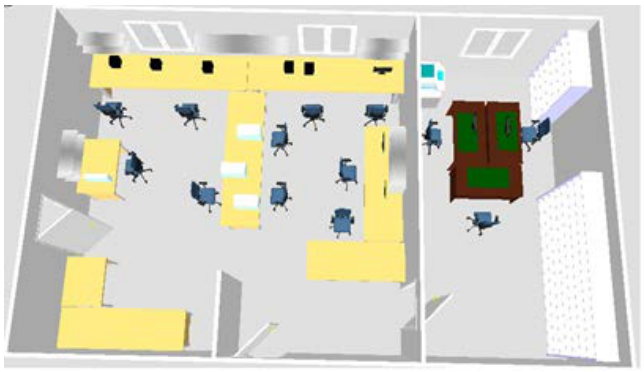
- KNX products must have a manufacturer ID (APP_MANUF_ID) that is assigned by the KNX association.
- KNX products have an application ID (APP_APPL_ID) managed by the product manufacturer (APP_MANUF_ID and APP_APPL_ID combine to create KNX_HW_TYPE, a unique 6-byte value that identifies the hardware).

The following two values are defined for keeping track of the application version and the ordering number to purchase the product:

- The version number (APP_APPL_VERSION) is managed by the product manufacturer.
- A 10-byte application order number (APP_ORDER_NR) can be specified. This value is not used by the system but can be displayed in ETS for installers to use in their projects.

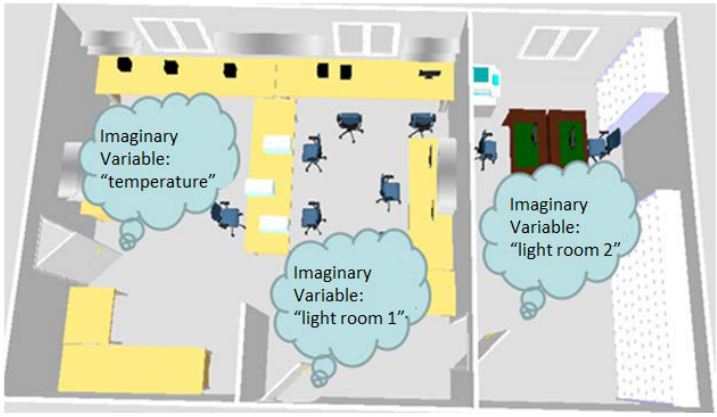
The required information for adding communication objects is defined. To understand communication objects, consider the office floor in Figure 30.

Figure 30. Typical Office Floor



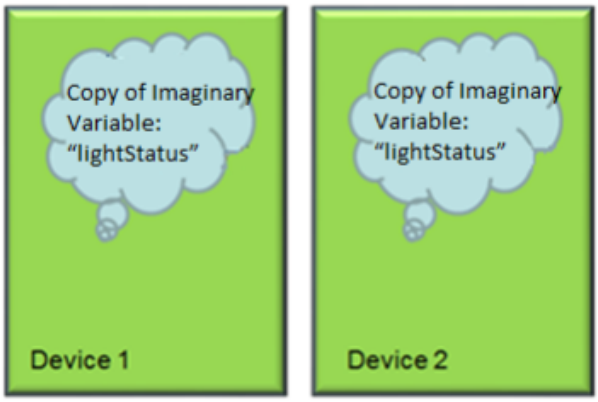
Each room in the office floor may have different sensors that represent the state of the room. Each sensor will have one imaginary variable that defines the state of that particular device, as Figure 31 shows. These variables are network variables.

Figure 31. Office Network Variables



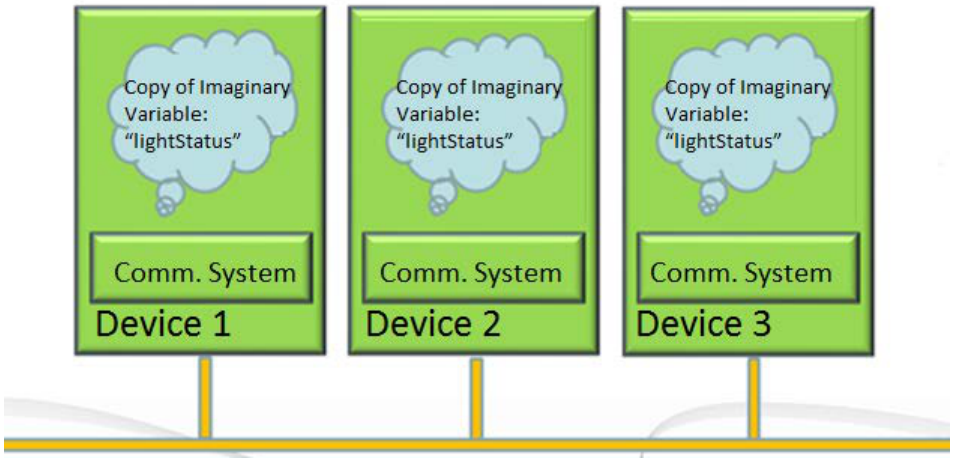
A device may have a copy of an specific imaginary variable. Several devices may have individual copies of the same imaginary variable as in Figure 32.

Figure 32. One or More Devices Having a Copy of *lightStatus* Imaginary Variable



The communication system keeps individual copies of the imaginary variable synchronized across devices as in Figure 33. These copies are communication objects.

Figure 33. Communication System Keeps Imaginary Variables Synchronized



Consider a room with five lightbulbs and one switch. When the switch is flipped, the lightbulbs turn on. Under this configuration, the five bulbs belong to the same group communication object and the switch belongs to a different communication object. Without grouping of objects, the switch would have to send five individual messages (one to each bulb). The total number of communication objects is stored in APP_objectTabSize. A system with four input objects and four output objects must define APP_objectTabSize equal to eight. The communication objects are also defined as part of the BCU-RAM. Figure 34 shows an example of a BCU-RAM structure for an input object (in0) and an output object (out0).

Figure 34. BCU-RAM Structure for an Input Object (in0) and an Output Object (out0)

```
typedef struct {
  unsigned char ramFlags[APP_objectTabSize];
  unsigned char in0;
  unsigned char out0;
} APP_Ram;
```

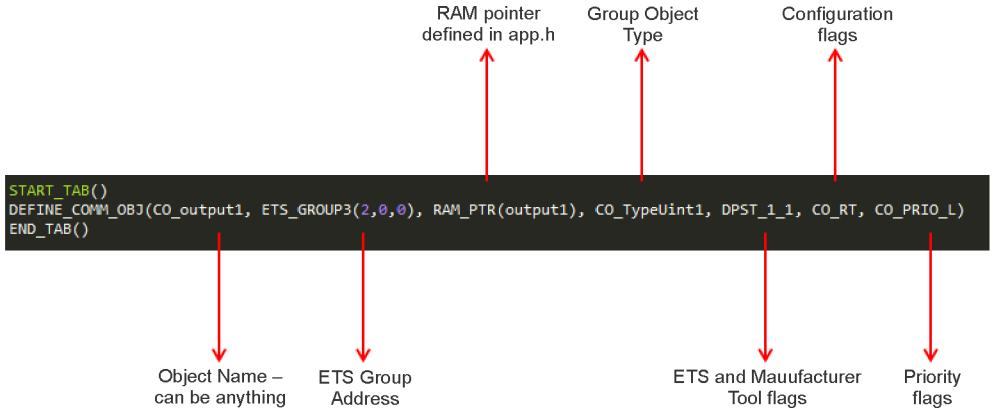
4.3.2.2 app_data.c

app_data.c includes header files for memory mapping. Generally, no changes are required in this file unless ETS parameters are defined in app.h.

4.3.2.3 cotab.h

This file serves two main purposes. The first is to enable the definition of a device individual address (KNX_DEF_INDIVIDUAL_ADDR). This address may be modified by ETS later during installation. The device address allows accessing the device in a peer-to-peer manner. The second important task of this file is to define the communication objects of the application. In app.h, the number and RAM variable names for the objects are defined. In cotab.h, the communication objects get a name, a group address, a pointer to the variables in the BCU-RAM structure, a data type, ETS flags, and configuration flags. Figure 35 shows an example. The name is used in the application to access the object and can be any string. The group address is defined following the convention of Figure 13 by three values: the main group, middle group, and subgroup.

Figure 35. Macro for Defining Communication Objects



The order of the macros defines the order of the group communication objects. There are restrictions on the associated addresses. There is a one-to-one assignment for group addresses to the group communication objects. The group addresses must be in ascending order. For example, ETS_GROUP (2,0,0) must precede ETS_GROUP (2,0,1) and follow ETS_GROUP (1,6,8). In this example, the group object type is defined as CO_TypeUint1 and can be used for any of the 1-bit unsigned data type such as the one Figure 14 shows. For a complete list of group object types and priority flags, see KAIstack API→Group communication→Reference in the KAIStack manual. The ETS & Manufacturer tool flags are data point types defined in the knx_master.xml file in the ETS installation file (typically \Program Files (x86)\ETS5\knx_master.xml). In this example, DPST_1_1 is used.

Figure 36 shows the definition of this flag as DPT_Switch in knx_master.xml. For a complete list of ETS and Manufacturer tool flags, see knx_master.xml.

Figure 36. knx_master.xml Definition of DPST_1_1 as DPT_Switch

```
<DatapointSubtype Id="DPST-1-1" Number="1" Name="DPT_Switch" Text="switch" Default="true">
  <Format>
    <Bit Id="DPST-1-1_F-1" Cleared="Off" Set="On" />
  </Format>
</DatapointSubtype>
```

The configuration flags define the type of operation (such as read only) of the communication object. Figure 37 shows a full list of flags.

Figure 37. Configuration Flags for Communication Objects

	CO_commEnable	CO_transmitEnable	CO_writeEnable	CO_readResponseEnable	CO_readEnable	
CO_T	X	X				Transmit only
CO_RT	X	X			X	Transmit, Read
CO_W	X		X			Write
CO_WU	X		X	X		Write, ReadResponseUpdate
CO_RWU	X		X	X	X	Write, ReadResponseUpdate, Read
CO_R	X				X	Read

4.3.2.4 main.c

After a reset, main is called. In main, the peripherals and stack are initialized and the software does cyclical calls to App_main().

4.3.2.5 project.h

This header file defines the device mode, the target compiler, the device derivative, the evaluation board information, and the stack size. AppWizard automatically populates this file based on the user-selected configuration.

4.3.2.6 bulb.c

The Appwizard names this file based on the project name. bulb.c contains the application functions in Figure 25 for initialization and runtime cyclical operation. bulb.c is where the user code is added for the application.

NOTE: The code examples that are included in the installation of KAIstack use functions that start with the characters KSD_. These functions used in the examples inside \appl_examples are defined in \system_15\targets\TI\msp430_common\KAIstackEval_MSP430xxx.

4.3.3 Adding an Application to the KNX Project

If the KNX application is a simple lightbulb that reads the status of a KNX switch, the lightbulb turns on or off whenever the status of the switch changes. If the lightbulb has a user-controlled status push-button, the status of the lightbulb is sent over the bus when the button is pressed. For this example, pin 1.1 is assumed to be connected to a push-button switch and pin 1.0 is assumed to be connected to a lightbulb (or LED). A typical project for this application would have three source files and two or three header files. Suppose you have the following files:

- init.c
- init.h
- app_main.c
- sensor.c
- sensor.h

init.c contains all the initialization functions required to run the application. Suppose void ioInit() is available. sensor.c has the function void bulbStatusUpdate (unsigned char status) that turns the bulb on or off.

app_main.c calls the initialization function from init.c and periodically checks if a KNX message from the switch is available. If a message has been received, the read value is compared to the previous reading and if any change is required on the lightbulb, the bulb changes state. If the switch has been pressed, the status of the bulb is sent over the bus. An interrupt is used for the user push-button. The header files have the function declarations required for the app_main.c to access the functions. Figure 38 shows pseudo code for the source files. TI recommends following the next steps by creating similar source and header files.

Figure 38. Pseudo Code for app_main.c, init.c, and sensor.c

app_main.c

```
#include "init.h"
#include "sensor.h"

int main(void) {

    ioInit();

    unsigned char sendStatus = NO;
    unsigned char bulbStatus = OFF;
    unsigned char bulbStatusNew;

    while(1){

        if message received over KNX bus
        {
            check new status of KNX_switch
            if bulbStatusNew != bulbStatus
            {
                bulbStatusUpdate(KNX_switch value)
            }
        }
        if(sendStatus equals YES)
        {
            sendStatus = NO;
            send a KNX message with the current status of the lightbulb
        }
    }
}

//*****************************************************************************
// Interrupt Service Routine - Port 1
//*****************************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    switch (__even_in_range(P1IV, P1IV_P1IFG7)) {

    case P1IV_NONE:   break;                    //No Interrupt pending
    case P1IV_P1IFG0: break;                    //P1IV P1IFG.0
    case P1IV_P1IFG1:                           //P1IV P1IFG.1 - Button
    // Clear P1.1 IFG
    P1IFG &= ~BIT1;
    // Set sendStatus to trigger a message on the bus
    sendStatus = YES;
    break;
    case P1IV_P1IFG2: break;                    //P1IV P1IFG.2
    case P1IV_P1IFG3: break;                    //P1IV P1IFG.3
    case P1IV_P1IFG4: break;                    //P1IV P1IFG.4
    case P1IV_P1IFG5: break;                    //P1IV P1IFG.5
    case P1IV_P1IFG6: break;                    //P1IV P1IFG.6
    case P1IV_P1IFG7: break;                    //P1IV P1IFG.7
    }

}
```

init.c

```
#include "init.h"

void ioInit(void) {
    // For FRAM devices, disable the GPIO power-on default high-impedance mode
    PM5CTL0 &= ~LOCKLPM5;

    // Alarm button
    // Direction is input
    P1DIR &= ~BIT1;
    // Internal pull-up on P3.4
    P1OUT |= BIT1;
    // Enable pull-up resistor
    P1REN |= BIT1;
    // Lo/Hi edge
    P1IES &= ~BIT1;
    // Clear all P3 interrupt flags
    P1IFG = 0;
    // interrupt enabled
    P1IE |= BIT1;

    // Output lightbulb
    // Direction is output
    P1DIR |= BIT0;
    // Initial state is OFF
    P1OUT &= ~BIT0;

}
```

sensor.c

```
#include "sensor.h"

void bulbStatusUpdate (unsigned char status) {

    if (status == ON)
        P1OUT |= BIT0;
    else
        P1OUT &= ~BIT0;

}
```

Figure 39 shows code for the header files. MSP430FR5969 is assumed for this example.

Figure 39. Code for sensor.h and init.h

sensor.h

```
#include "msp430fr5969.h"
#define     ON      1
#define     OFF     0


void bulbStatusUpdate (unsigned char status);
```

init.h

```
#include "msp430fr5969.h"
#define     YES     1
#define     NO      0

void ioInit(void);
```

4.3.3.1 Configuring KNX Files

To add the application code to the KNX project, configure the KAIstack files in Section 4.3.2. For the development stage, any numbers can be given for the IDs of app.h. The important parameters for testing the system are as follows:

- The address table size
- The association table size
- The communication object size
- The BCU-RAM structure

APP_addrTabSize and APP_assocTabSize depend on the device model and the number of communication objects in the application. The device model establishes the maximum number of group addresses that can be allocated. Figure 40 shows the maximum numbers for device model 0705. For specific device model information, see the KAIstack reference manual.

Figure 40. Example of Maximum Group Addresses for Device Model 0705

Device Model 0705 Overview

Overview

Features

- Maximum number of group addresses: 252
- Maximum number of associations: 254
- Maximum number of group objects: 254

The maximum number of memory allocation for the tables is not always required. If four communication objects are required for the application, reserving 254 addresses statically wastes resources. For four communication objects, tables of size equal to four is sufficient. The simple switch example has only two communication objects: an input to receive messages from the bus and an output to send messages on the bus. APP_objectTabSize must equal 2 and the most efficient size for the address tables is also 2. Figure 41 shows the configuration.

Figure 41. Communication Tables Setup in app.h

```
//*****************************************************************************
// Size of system tables
//*****************************************************************************

// Size of the address table (excluding physical address)
// With this constant the amount of memory for the address table is reserved.
// The maximum allowed size depends on the device model.
#define APP_addrTabSize      2

// Size of the association table
// With this constant the amount of memory for the association table is reserved.
// The maximum size depends on the device model.
#define APP_assocTabSize     2

// Number of group communication objects.
// The maximum size depends on the device model.
// For this example we have 1 output and 1 input,
// so the total number of objects is 2
#define APP_objectTabSize    2
```

The BCU-RAM structure must be defined to include the communication objects as in Figure 42. The output communication object is statusOut and the input communication object is statusIn. The name is inconsequential but is used in the application to modify the value of the communication object and send it over the bus.

Figure 42. BCU-RAM Includes the Name of the Output Communication Object

```
//*****************************************************************************
// Definition of BCU-RAM
//*****************************************************************************
typedef struct {
  unsigned char ramFlags[APP_objectTabSize];            // RAM-flags
  unsigned char statusIn;                                    // object value: input
  unsigned char statusOut;                                   // object value: output
} APP_Ram;
```

NOTE: To use C99 types (such as uint8_t), stdint.h must be included (#include<stdint.h>) in app.h. This file is not included by default. Also, app_make.gmic in \workspace\gmake must have the following line:

ADD_INCLUDE_PATH += -I $(PATH_APPL)/../../../Compiler/MSP430_IAR6/inc/dlib/c/

See Figure 43.

Figure 43. ADD_INCLUDE_PATH to Point Compiler to Correct stdint.h File

```
SOURCES_APPL_BASIC = $(PATH_APPL)/bulb.c \
                     $(PATH_APPL)/app_data.c \

SOURCES_APPL_FIX =  $(SOURCES_APPL_BASIC) \
                    $(PATH_APPL)/main.c \

SOURCES_APPL_ETS  = $(PATH_APPL)/app_data.c \

DEPENDENCIES_APPL = $(wildcard $(PATH_APPL)/*.h)

ADD_INCLUDE_PATH += -I $(PATH_APPL)/../../../Compiler/MSP430_IAR6/inc/dlib/c/
```

cotab.h requires the following changes.

1. Make KNX_CURRENT_ADDR_TAB_LEN and KNX_CURRENT_ASSOC_TAB_LEN equal to the number of communication objects in the system (2 as in Figure 44).

Figure 44. Virtual Address Length Based on the Number of Communication Objects in cotab.h

```
/**
defines the current length of the address table
@see APP_addrTabSize
*/
#define KNX_CURRENT_ADDR_TAB_LEN 2

/*'
defines the current length of the association table
@see APP_assocTabSize
*/
#define KNX_CURRENT_ASSOC_TAB_LEN 2
```

2. Declare the communication objects.

In this case, two 1-bit objects are required: one for input and one for output. Figure 45 shows the code for a configuring statusIn as a 1-bit low-priority input object with a group address of 2/0/0 and statusOut as a 1-bit low-priority output object with a group address of 2/0/1. The names of the communication objects in the RAM-BCU structure of app.h are used as a parameter of RAM_PTR for the declaration of the communication objects.

Figure 45. Declaration of Output Communication Object

```
START_TAB()
DEFINE_COMM_OBJ(CO_statusIn,  ETS_GROUP3(2,0,0), RAM_PTR(statusIn), CO_TypeUint1, DPT_1, CO_RWU, CO_PRIO_L)
DEFINE_COMM_OBJ(CO_statusOut, ETS_GROUP3(2,0,1), RAM_PTR(statusOut), CO_TypeUint1, DPST_1_1, CO_RT, CO_PRIO_L)
END_TAB()
```

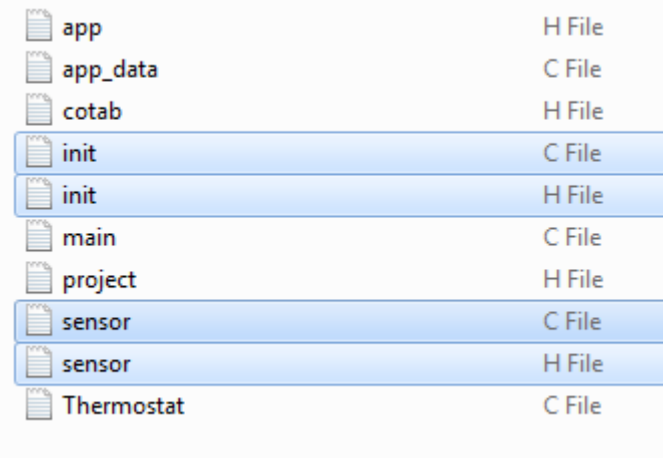
No changes are required in app_data.c, main.c or project.h.

4.3.3.2 Adding Application Files

The KNX project is ready to be integrated with the application code. To integrate this project with the application code, do as follows:

1. Copy all application files except app_main.c into the KNX \scr directory as in Figure 46.

Figure 46. Copy Application Files in to the \src Directory of the KNX Project

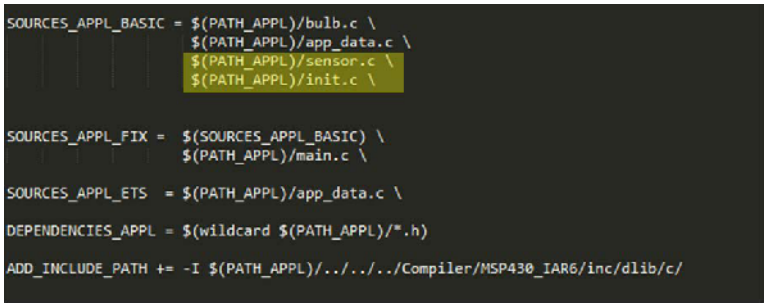


NOTE: Placing the files in this directory is insufficient for the compiler to find them.

2. Modify the file app_make.gmic in \workspace\gmake to update SOURCES_APPL_BASIC to include these files in the compiler search path as in Figure 47.

NOTE: init.c and sensor.c were added. Source files of the application must be listed on this file.

Figure 47. Add Source Files to app_make.gmic



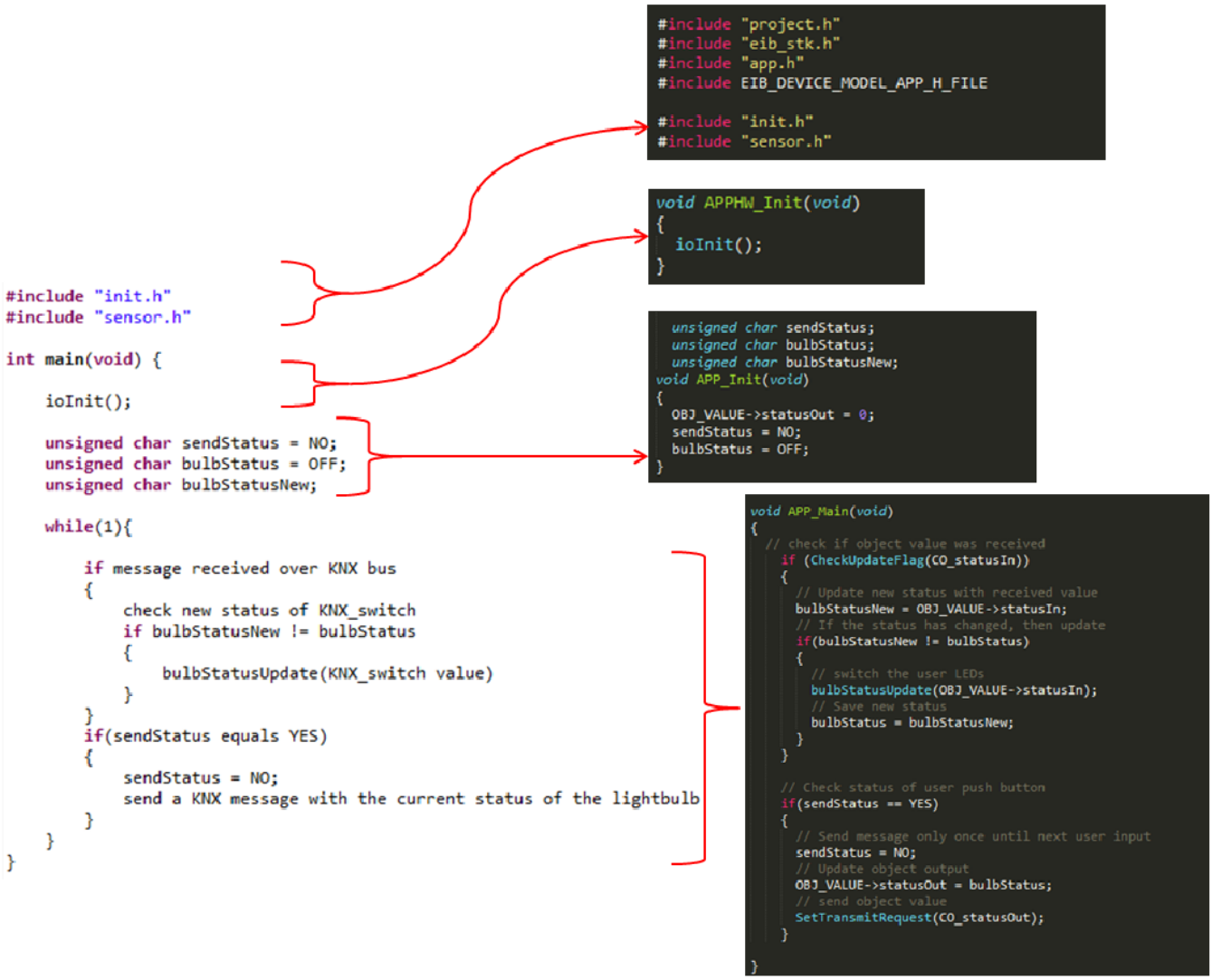
3. Add app_main.c to the KNX project (see Figure 48).
   (a) Include the header files from app_main.c to bulb.c to ensure that function calls in bulb.c are declared in the respective header files.
   (b) Place hardware initialization function calls inside APPHW_Init().

   NOTE: initIO() must be declared in init.h.

   (c) Add variables from app_main.c to bulb.c.
   (d) Initialize the variables and communication objects in APP_Init().
   (e) Add the cyclic code into APP_main().

When an input communication object is modified by a message on the KNX bus, a flag is set. To check for the flag, the stack provides the CheckUpdateFlag function that takes the input object as its argument. If the object has changed since the last check, the function returns TRUE. Otherwise, the function returns FALSE. An if statement can be used to determine whether a message for that input object has arrived. The value of the input communication object can then be read by accessing the specific BCU-RAM value (readInput = OBJ_VALUE → inputObject). If a message must be sent on the bus, the output communication object statusOut is modified (OBJ_VALUE → statusOut = value). The function SetTransmitRequest() sends a message on the KNX bus. The parameter passed to SetTransmitRequest() is the object CO_statusOut declared in cotab.h and not statusOut.

Figure 48. Adding app_main.c to bulb.c



The project is ready for compiling. Before closing this section, note two additional aspects of bulb.c. The first is the App_Save() function. Whenever there is a power down on the KNX bus, the PHY board notifies the MSP device of the event through an IO signal. When this signal is received, KAIstack calls the APP_Save function. APP_Save must contain critical code to be executed before the device shuts off. An example is saving a critical variable to memory. The second point is the use of interrupts. Interrupts can be used as in any other MSP application by declaring the interrupt service routine (ISR) in bulb.c.

Figure 49 shows an example of such declaration for the port interrupt described in this example.

Figure 49. Port 1 ISR Declaration in bulb.c

```
//*****************************************************************************
// Interrupt Service Routine - Port 1
//*****************************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    switch (__even_in_range(P1IV, P1IV_P1IFG7)) {

    case P1IV_NONE:   break;                    //No Interrupt pending
    case P1IV_P1IFG0: break;                    //P1IV P1IFG.0
    case P1IV_P1IFG1:                          //P1IV P1IFG.1 - Button
    // Clear P1.1 IFG
    P1IFG &= ~BIT1;
    // Set sendStatus to trigger a message on the bus
    sendStatus = YES;
    break;
    case P1IV_P1IFG2: break;                    //P1IV P1IFG.2
    case P1IV_P1IFG3: break;                    //P1IV P1IFG.3
    case P1IV_P1IFG4: break;                    //P1IV P1IFG.4
    case P1IV_P1IFG5: break;                    //P1IV P1IFG.5
    case P1IV_P1IFG6: break;                    //P1IV P1IFG.6
    case P1IV_P1IFG7: break;                    //P1IV P1IFG.7
    }

}
```

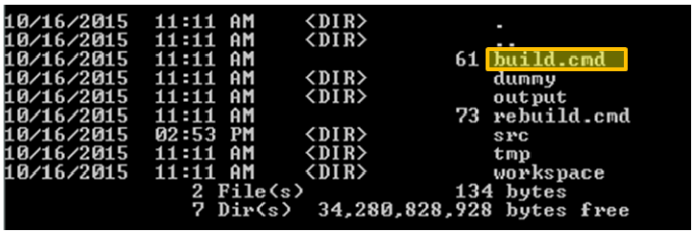
4.4 Compiling a KNX Project

KNX projects cannot be built in an IDE such as IAR or CCS. To compile a KNX project, do as follows:

1. Use the command file generated by the AppWizard to build the project.
2. Choose from the following options:
   - Option 1: Click build.cmd or rebuild.cmd.
   - Option 2: Execute from the command file from the file explorer.
     (a) Go to the project directory (\knx_app in this example) to open a command line in Windows.
     (b) Right-click the bulb directory while holding the Shift key.
     (c) Select *Open command window here*.
     (d) Type dir.
     (e) Press Enter to ensure the command line is in the correct directory.
   - Ensure that build.cmd is listed on command line as in Figure 50 (if not, you are in the incorrect folder).

NOTE: This action lists all files and directories in the current folder.

Figure 50. Finding the Build Program Inside bulb



In the correct directory, build the project as follows:

(A) Type *build rebuild*.

(B) Press enter.

If any error or warnings occur, they are listed. If the build completes successfully, the command line shows zero errors (see Figure 51) and the debug file bulb.d43 is created inside \output as in Figure 52.

Figure 51. Successful Finish of Build

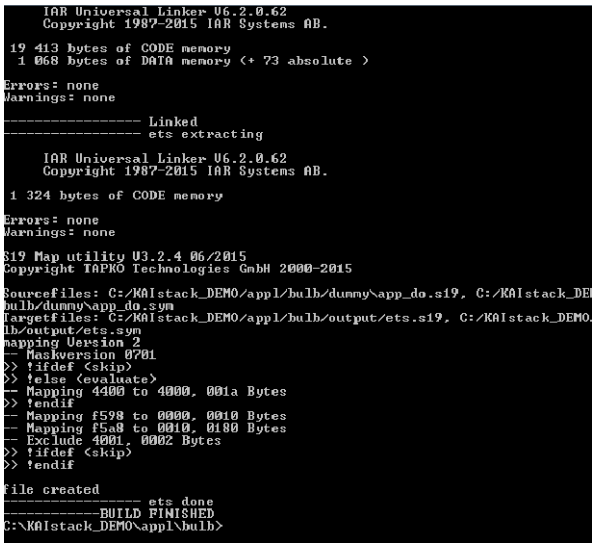


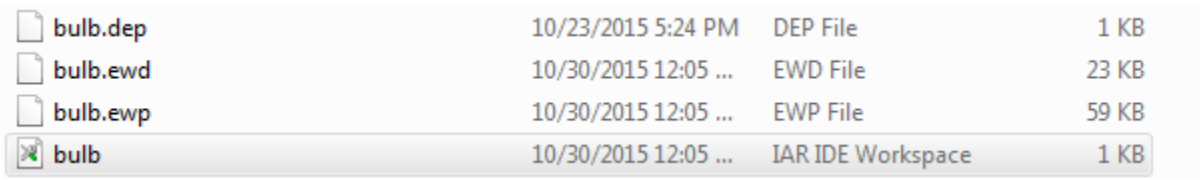
Figure 52. Generated Files in \output After a Project Build

The .d43 file is for programming and debugging the device.

4.5 Downloading and Debugging a KNX-Enabled Application

When the debug file (.d43) is generated, it can be downloaded to the memory of the device and debugged in the IAR environment. This section describes the steps required to program and debug the KNX project. Figure 53 shows that the AppWizard created a IAR IDE Workspace file (.eww) in \bulb\workspace\iar.

Figure 53. IAR Worspace File Inside \workspace\iar

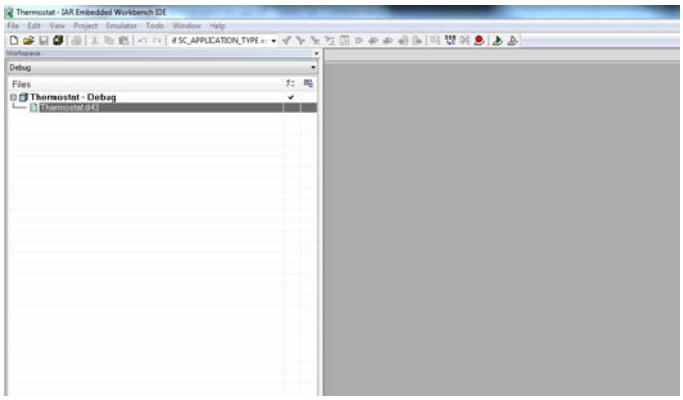


To program and debug the KNX project, open the IAR IDE Workspace file in Figure 53.

NOTE: This action launches IAR and loads the debug (.d43) file as in Figure 54. If this is the first time IAR is used, you may need to indicate to Windows to use IAR to open the .eww file. To open the .eww file, do as follows:

1. Right-click the file.
2. Select *Open with…*
3. Browse to the IAR executable on the PC.

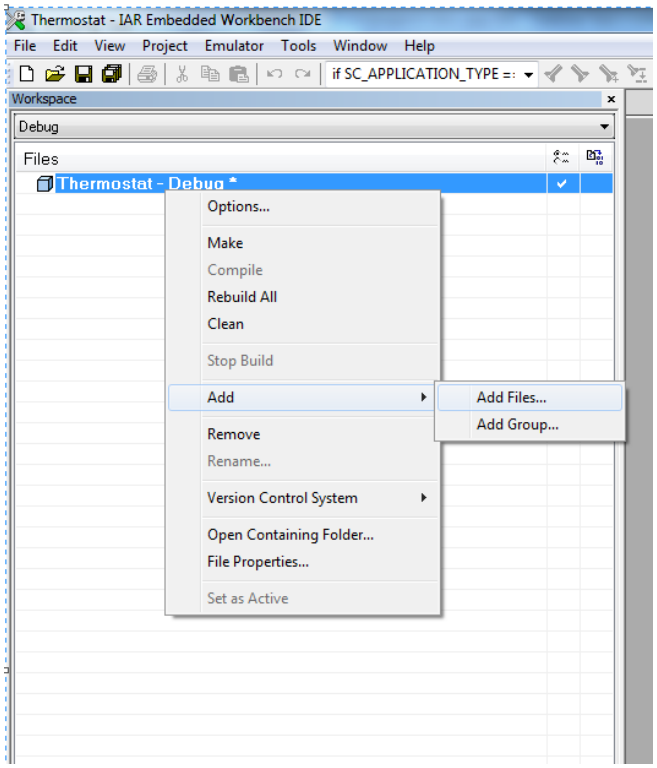
Figure 54. IAR Workspace With Debug File



If the debug file is not loaded by default or if you want to load a different file, do as follows.

1. Right-click on the name of the project.
2. Select *Add*.
3. Select *Add Files* as in Figure 55 which opens Windows Wxplorer.

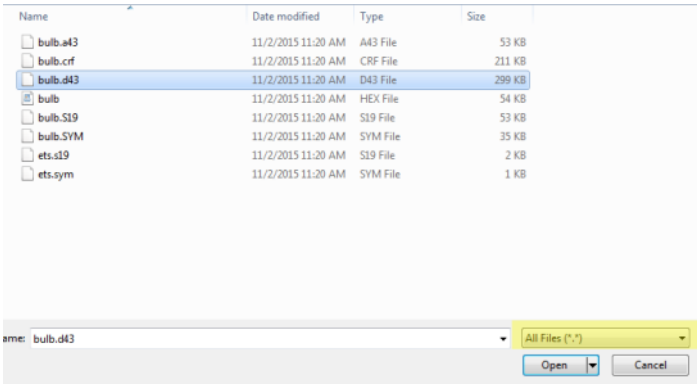
Figure 55. Adding a Debug File to the Workspace



4. Navigate to \output.
5. Select the debug file.

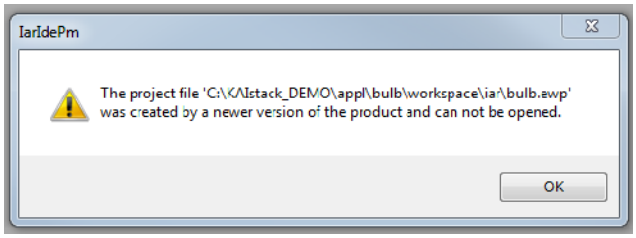
NOTE: The file type must be set to *All files* to show the.d43 file in this window as in Figure 56.

Figure 56. Manually Adding a Debug File to the IAR Workspace



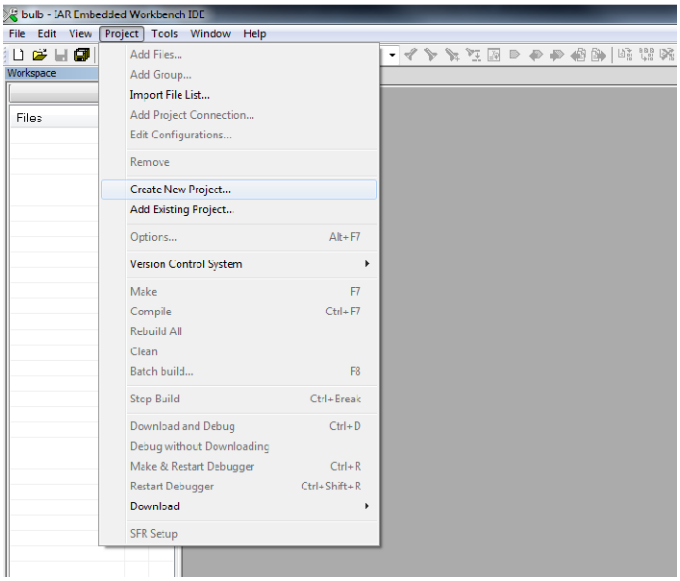
If the version of IAR installed is not the latest version, the window shows the error in Figure 57. To create a new project and add the files in the version of IAR you are running, do as follows:

Figure 57. IAR Open Workspace Error Due to Version Control



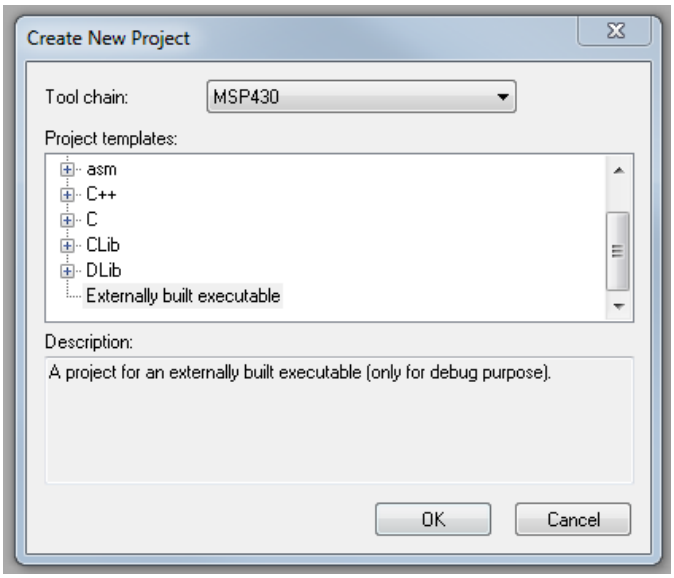
1. Click OK in the error message.
2. Click *Project*.
3. Click *Create New Project* as in Figure 58.

Figure 58. Create New Project in IAR



NOTE: A new project creation window opens as in Figure 59.

Figure 59. Create New Project Window



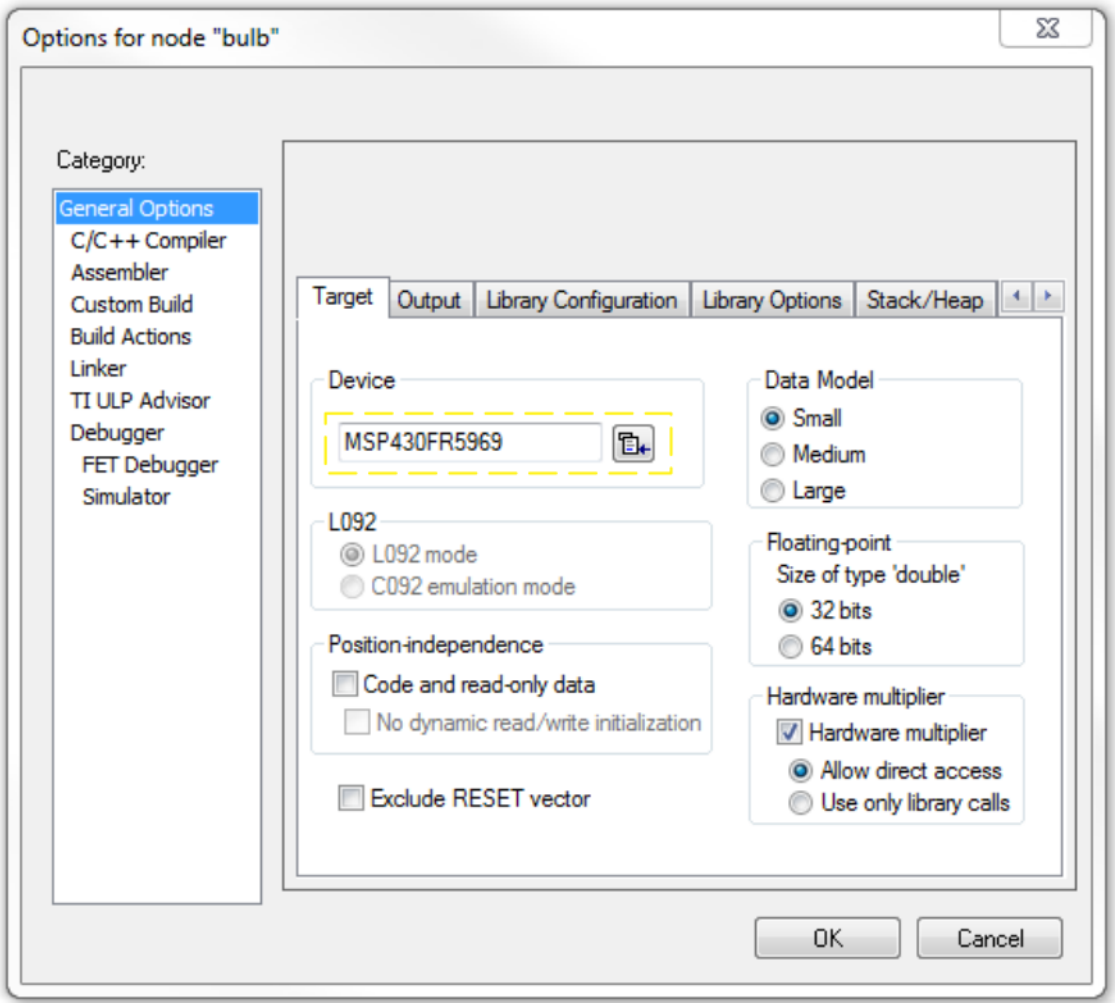
4. Select MSP430 as the tool chain.
5. Scroll to *Externally build executables* for the Project template.
6. Click OK to save.
7. Save the project anywhere on the computer.

The project must be configured for the specific device to be programmed. To configure the project for a specific device, do as follows:

1. Right-click on the project file in IAR (with the blue cube).
2. Select *Options*….

NOTE: This action opens the options windows in Figure 60.

Figure 60. Selecting the Settings of the Project



3. Navigate to *General Options*.
4. Select the correct device.

By default, the debugger in IAR is set to a simulator. Change this setting or the code will not download to the MSP memory. To change this setting, do as follows:

1. Click the *Debugger* tab.
2. Select *FET Debugger* as the driver.
3. Click OK.

If the error in Figure 61 occurs after the *Download and Debug* button is pressed, the selected device for the project is not the same as the hardware.

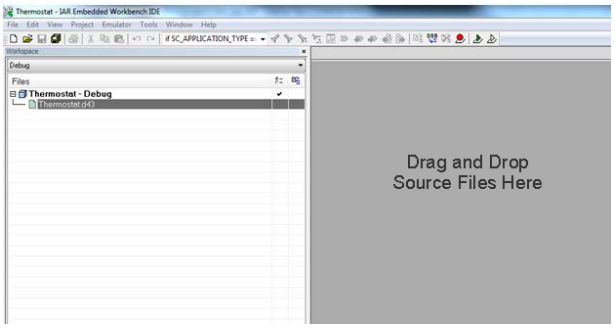
Figure 61. Error Due to Mismatch Between Hardware and Selected Device in IAR

The project is ready.

1. Add the debug file as in Figure 55.

NOTE: Before downloading the code, TI recommends opening all source files in IAR. Do not to add the source files to the workspace. Do not click *Add Files* to open the source code in IAR. To open the files, drag the files from the Window explorer to the IAR dark gray window in Figure 62. After opening the files in IAR and downloading the code to the device, you can place breakpoints, watch variables, memory, and register to take full advantage of the IAR debugging tools.

Figure 62. Add Source Files for Debugging



2. Click the *Download and Debug* button in Figure 63 to download the code.

Figure 63. Button to Download Code to Device and Begin Debug Session

You can change the source files in the IAR environment but you must rebuild the project in the Windows command line and refain from clicking the build option in IAR.

4.6 Stack Device Resources

KAIstack requires hardware and software resources to function. The designer must ensure that stack resources are not modified by the application firmware. For a full list of hardware resources used by KAIstack for a specific MSP, search for MSP430_TF (for TP-UART) or MSP430_DF (for bit-based PHY) in the KAIstack manual.

The TP-UART (KAIlink UART) configuration requires a UART channel and a couple of IO lines. The bit-based PHY version (KAIlink bit) uses a timer, IO lines, and the NMI pin. The bit-based PHY requires an external 16-MHz crystal. The footprint of the communication stack decreases the memory available on the MSP. The actual memory requirements depend on the chosen configurations of the stack.

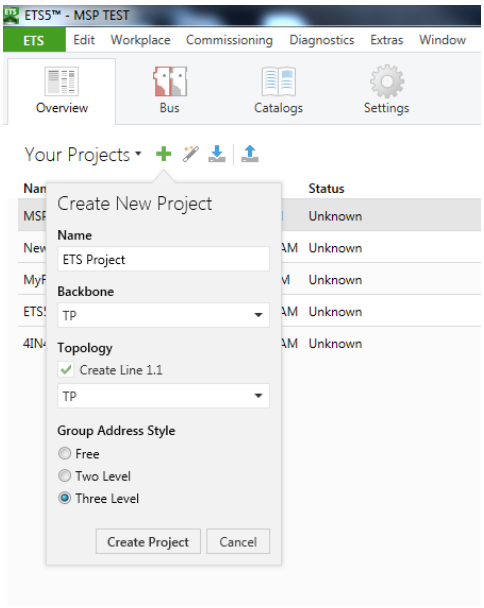
The stack uses approximately 20KB of nonvolatile memory and 1.5KB of volatile memory. The stack configuration controls the core frequency. Frequency options for each MSP device can be found in the KAIstack manual.

4.7 Testing the KNX Project With ETS

After downloading the firmware to the device using IAR, you can use ETS to ensure that the communication is operating. In addition to the MSP board and the PHY hardware, a KNX power supply and the USB-KNX interface are required for testing. To use ETS to ensure that communication is operating, do as follows:

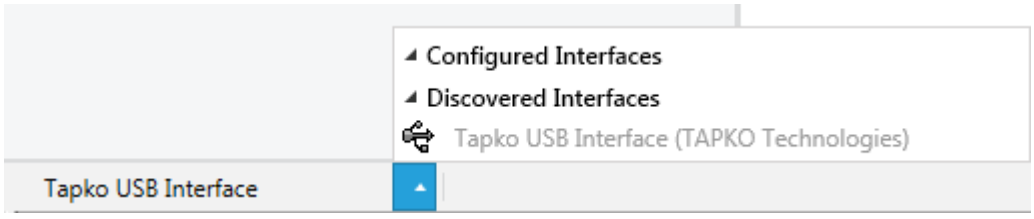
1. Connect the USB-KNX interface to the power supply, the application hardware, and the PC.
2. Open ETS to create a new project as in Figure 64.

Figure 64. Creating a New ETS Project



3. Open the main ETS window.
4. Ensure that the Tapko USB Interface is selected in the bottom left corner of the window as in Figure 65.

Figure 65. Ensure Tapko USB Interface is Selected in ETS



5. Click *Diagnostics*.
6. Select *Bus Monitoring*.

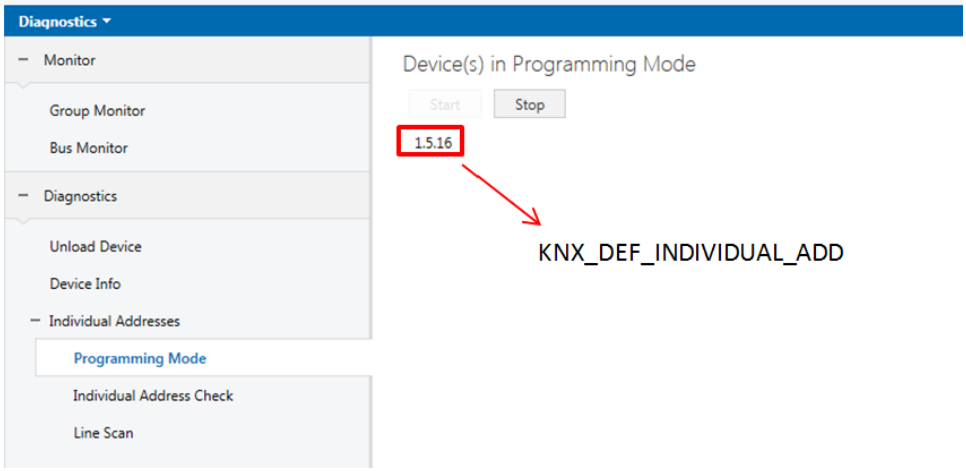
NOTE: This step opens a diagnostic window in which all receiving KNX messages can be seen. Messages can be sent to specific communication objects from this window.

Checking for devices in programming mode ensures the stack is executing correctly. In KNX, the programming mode can be used to change the individual address of the device from ETS. Enable programming mode to ensure the detection of the device on the bus for testing.

1. Select *Programming Mode* under *Individual Addresses* in the Diagnostics window.
2. Click *Start*.
3. Check the hardware documentation for your specific board for the programming mode button or switch to put the device in programming mode.

NOTE: The device address (KNX_DEF_INDIVIDUAL_ADDR) declared in cotab.h shows in the Diagnostics window if the hardware is in programming mode and everything is running correctly (see Figure 66).

Figure 66. Individual Address of the Device in Programming Mode



NOTE: If the individual address of the device appears on the Diagnostics window, the stack is executing and the communication system is correctly set up.

4. Click *Stop*.
5. Continue with the next test.

NOTE: If the address is not shown in programming mode, check the hardware for connections. Software debugging may be required.

Check if messages can be received on the PC from the device. The lightbulb application has the communication objects in Figure 67. One input object with group address 2/0/0 and one output object with group address 2/0/1 are declared.

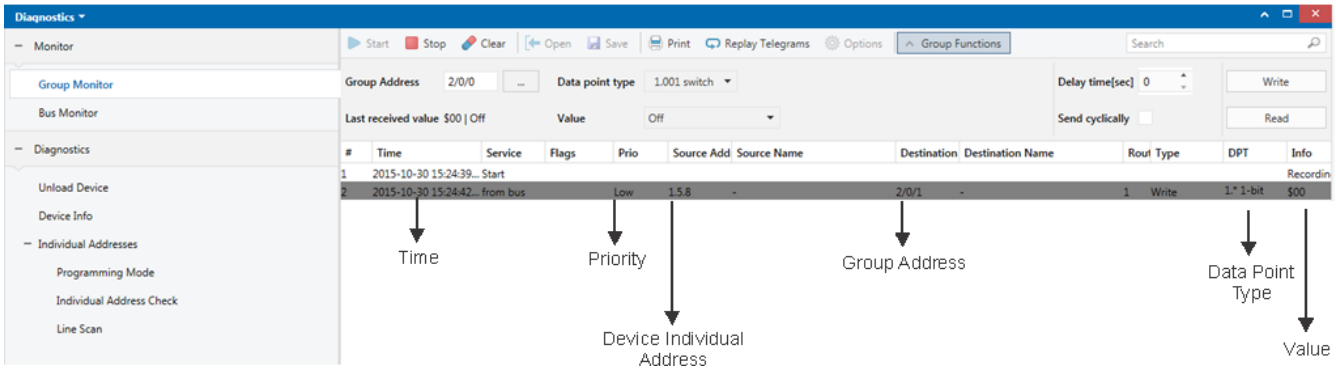
Figure 67. Communication Objects for Testing With ETS

```
START_TAB()
DEFINE_COMM_OBJ(CO_statusIn,  ETS_GROUP3(2,0,0), RAM_PTR(statusIn), CO_TypeUint1, DPT_1, CO_RWU, CO_PRIO_L)
DEFINE_COMM_OBJ(CO_statusOut, ETS_GROUP3(2,0,1), RAM_PTR(statusOut), CO_TypeUint1, DPST_1_1, CO_RT, CO_PRIO_L)
END_TAB()
```

1. Select *Group Monitor* under *Monitor* in the Diagnostic window to see the KNX messages sent by the device over the bus.
2. Click *Start*.
3. Press the button on the hardware (in our example, pin 1.0) to send a message on the bus.

NOTE: Figure 68 shows the complete information including the time, priority, individual source address, output object address, data point type, and value in the displayed information. In this case, the output object with group address 2/0/1 (CO_statusOut in Figure 67) sent a message of 1 bit with a value equal to 0.

Figure 68. Messages Seen by ETS From Output Communication Objects

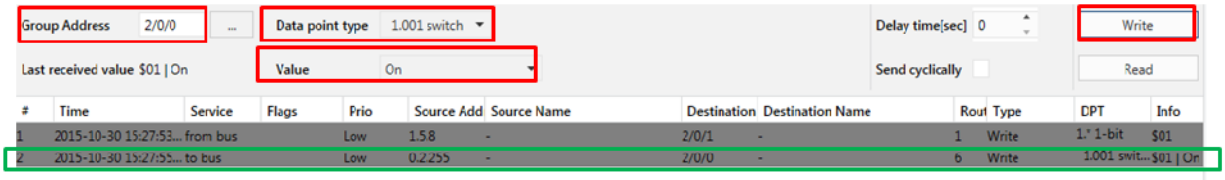


Messages can be sent from ETS to the input communication object declared in Figure 67. The message must be configured. To configure the message, do as follows:

1. Select the group address of the input object (in this case, the input has a group address of 2/0/0).
2. Select the data point type (the input in this example is of data point type CO_TypeUint1, which is a 1-bit value, so the 1.001 switch option is selected [any 1-bit option works in this step]).
3. Select either 0 (Off) or 1 (On) in the *Value* field.
4. Click *Write*.

NOTE: ETS sends a message to group address 2/0/0 of 1 bit with this value. Pin 1.1 changes based on the value written by ETS. Figure 69 shows the configuration.

Figure 69. Sending a Message to Input Object With Group Address 2/0/0 From ETS



To check for message reception on the MSP application, check the flag for statusIn is checked in APP_Main() as in Figure 70. Put a breakpoint inside if statement to debug the reception of messages.

Figure 70. Check for Change in Input Communication Object Value

```
void APP_Main(void)
{
  // check if object value was received
    if (CheckUpdateFlag(CO_statusIn))
    {
      // Update new status with received value
      bulbStatusNew = OBJ_VALUE->statusIn;
      // If the status has changed, then update
      if(bulbStatusNew != bulbStatus)
      {
        // switch the user LEDs
        bulbStatusUpdate(OBJ_VALUE->statusIn);
        // Save new status
        bulbStatus = bulbStatusNew;
      }
    }
```

5 References

1. *KNX Association website* (www.KNX.org)
2. *KNX Basic Course Documentation*, December 2013.
3. *KNX Advanced Course Documentation*, November 2013.
4. *Tapko Technologies GmbH website*, (www.tapko.de)
5. *Serial Data Transmission and KNX Protocol*, KNX Association, http://www.knx.org/fileadmin/template/documents/downloads_support_menu/KNX_tutor_seminar_page/tutor_documentation/05_Serial%20Data%20Transmission_E0808f.pdf
6. *Interworking*, KNX Association, http://www.knx.org/fileadmin/template/documents/downloads_support_menu/KNX_tutor_seminar_page/Advanced_documentation/05_Interworking_E1209.pdf
7. *ETS5 for Beginners*, KNX Association, http://www.knx.org/media/docs/Flyers/ETS5-For-Beginners/ETS5-For-Beginners_en.pdf
8. *How to become n ETS App Developer*, KNX Association, http://www.knx.org/media/docs/Flyers/How-to-Become-An-ETS-App-Developer/How-To-Become-An-ETS-App-Developer_en.pdf
9. *KNX Development Getting Started – KNX System Components*, KNX Association, http://www.knx.org/media/docs/Flyers/KNX-Development-Getting-Started/KNX-Development-Getting-Started_en.pdf
10. *Texas Instruments ISO 9001 Certification*, http://focus.ti.com/pdfs/qlty/TI_286691_ISO9001_Final_Certificate.pdf
11. *The worldwide STANDARD for home and building control*, KNX Association, http://www.knx.org/media/docs/Flyers/KNX-Introduction-Flyer/KNX-Introduction-Flyer_en.pdf

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (https:www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265