KNX Runtime Engine

Table of Contents

|  |  |
| --- | --- |
| Pg. | Section |
|  | About |
|  | Components |
|  | Nodes |
|  | Types |
|  | Keywords |
|  | Bytecode |
|  | Scripts |
|  | Operators |
|  | Option Strings |
|  | Libraries / Modules |
|  | Command Line Options |
|  | Versioning |

About

The KNX language has been development for several years, frequently being written, shelved, scrapped and rewritten again. Each time being adjusted, improved, ignored, scrapped and rewritten again.

Hopefully this time it sticks.

Made a whole UML diagram and everything.

The intent of this language is to provide a simple means for creating multithreaded applications, in which individual interpreting threads, or ‘nodes’ as the kids say, to allow for granular delegation of duties for each part of a program.

As multithreading is an integral part of the design of the language, care has been taken to remove the burden of resource management and thread safety from the user to the runtime-time environment.

KNX is designed to be primarily used in prototyping, simple game design, communications, simulations and other such applications that normally take taxing periods of time and require complex processing.

Provided in this repository is the source code for all components and standard libraries, as well as an SDK (aka KDK) for use in the creation of custom libraries and modules to further benefit individual users. Please see the system documentation sections below, as well as the provided MIT license. It’s like two paragraphs, read it.

Components

If you are not interested in improving or maintaining the source code of this project, or in creating custom modules, you may skip this section as it contains primarily technical details of the design and workings between modules.

In following section, each modular component of the library will be broken down into its purpose, implementation, interfaces, data structures, and finally its interaction with other such modules.

**RTE**

The front facing portion of the KNX interpreter is the RTE, or **r**un-**t**ime **e**ngine. This acts a mediator between all other sections besides the SDK, sharing resources as needed, making calls to each other section of the program, as well as interacting with the user. The RTE performs nearly no data transforms, and simply acts a way to keep state and ensure components are compatible and work together properly.

**Components**

**RTE**

Like most other modules in this list, the RTE generally adopts its critical components from the SDK, so that they may be shared across different modules easily via the provided interfaces. However, there are a few structures and methods that are exclusive to the RTE.

Most important of these is the **Config** structure. This globally defined static variable contains user defined options, as well as some set at compilation time via macros, that allows a user to adjust how the runtime engine will operate. See the “Command Line Options” section for more details.

This brings us into the next component of the RTE, the command-line interpreter. This accepts options passed to it proceeded by dashes to flag certain operations, such as loading of scripts or printing of certain information, as well as setting basic configurations, such as thread and memory limitations.

The runtime engine component is fairly simple; it passes through a short number of phases to prepare the system, and then launches and hosts the root node (node 0). This allows the program to end when the root node is terminated, and safely shut down all active threads. The procedure of the RTE is as below:

* Initialize core components
* Parse command line operations
* Register and launch root node
* Close down all modules and components

One the root node has taken over, the RTE remains as the root processes of the program and only awaits shutdown.

**KDK**

One of the most important parts of the KNX ecosystem is the KDK, or **K**NX **D**evelopment **K**it. This provides a common set of structures and interfaces that allows for compatibility between modules, as well as third party modules. The KDK is necessary for creating custom c libraries importable to the engine. As such, it is important that this be kept up to date to prevent crashes or data corruption. This is updated along-side all other components, so as long as all are updated at once (and why wouldn’t they be?) there should be no problems stemming from this coupling of components. The use of this SDK allows for more rapid and modular development between components so updates will require only the specific modules that were changed between versions.

Remember; component versions are listed separately. If for some reason a component’s binary is incompatible (say, a component was upgraded via a USB copy), both the component version AND the build number are available via the --version command line option. The interpreter will also display a warning upon launch if a component is out of date, assuming nothing crashes during module load.

To go further into detail about the assets provided through the KDK, common structures will first be noted. As the KDK is the largest piece of the program, it is likely to take a while to fully delve into each piece and its purpose.

*Structures*

* ComponentInfo: Used in all components to supply version information, as well as other metadata.
  + Field: \_version; Version #stores version information
  + Method: GetVersion; Version #version information {get}
* Object: The most primitive data type. This acts a wrapper to a piece of data and supplies meta information about it, as well a type index that allows its type and casting rules to be ascertained as needed. Objects are storable in memory trees and other data registries.
  + Field: \_typeIndex; int #registered type index
  + Field: \_hash; ULONG #quick identifier
  + Field: Data; void \* #generic info pointer
  + Method: GetType(): ULONG #quick identifier {get}
  + Virtual Method: Constructor(); IMethod #Overridable type factory
* MemDb: Implements InfoStructure. This is used for object data storage.
  + Field:

//TODO finish when fleshed out

**DataManager**

The **Data Manager** is responsible for **object** registration and lookup. When registered, it becomes possible to determine appropriate constructors and conversion protocols. Using the KDK, it is possible to register custom data types and non-class methods upon loading the module.

When a new module is loaded, it may use the IObjectRegister interface for registering a new object type, or the IMethodRegister interface for methods. In the method case, it is important to use the registry to resolve type ids for the argument list expected in the method. If this requires another module to be loaded, this may also be requested. If the required libraries are not found, the module will fail to load. If a type required comes from the same library, ensure that it is registered first. Be aware, however, that as of this version if a library is unloaded, its dependencies will remain. This means that the dependency library will stay loaded. This fix will be slated for the future. Unless it is figured out before release, anyway.

An important note that should be made is that the **data manager** is NOT a memory bank; this exists only to store methods and object templates. All instances are stored by module. In the event of a symbol lookup collision, the first library on record will be used, unless an explicit namespace is stated or implicitly declared via the **implicit** operator.

**Module Manager**

To allow for binary libraries and plug-ins, the module manager provides a means to load dynamic libraries placed in either the *libs/stdlib* or *lib/extlib* directories. Note that any third-party or custom libraries should be placed in *extlib*.

A simple component, the module manager allows for the import, refresh and unload of a library made with the KDK. This allows users to import libraries which contains new data types, methods and other implementations. Copies of these libraries are cached in the *module register*, which keeps track of the number of instances a library has been imported, and tracks which nodes are in possession of an instance. In this way, multiple nodes may import a library, but it needs only to be loaded once. When all nodes currently using the library have released their instances, the library is unloaded from the register.

**Parser**

Of course, the primary goal of this software is to analyze, interpret, and finally execute code. The parser deals with plain-text code to convert it into usable bytecode, saved in a linked list of tokens. These tokens contain additional metadata, such as the data type of each word, as well as the raw text that symbols or literals where derived from. The parser also ensures that multi-line or nested statements are buffered until completion.

The parser uses a modified version of the shunting-yard algorithm to arrange command phrases, even when nested or containing lists. This is done by the use of a stack array of function tokens, as well as an array of output tokens that preserve function body bounds and allow for lists of values to be stored on a single row. If a group has not been completed, it is stored in a buffer and awaits further raw text. When all encapsulations have been closed and the parser is no longer awaiting further input, the executor is then allowed to run the tokenized data.

**Executor**

Executing the token sequence is done by iterating to the first function statement. Depending on the operator, between 0 and 2 of the previous value inputs are passed as arguments to the selected operation. Most mathematical operations accept one or two arguments, although in cases such as the root operator (‘√’) the number of arguments accepted may vary. In this case, the parser will detect an implied value and substitute it automatically.

When a statement is executed, it is then substituted with a token representing the return value. Reading continues normally from this point. If the node in question is enabled for user input, operation will halt until further commands are manually issued. Otherwise, nodes will simply wait for new commands to appear in their stack.

Nodes

Nodes are central to the design and operation of the KNX language. To define a node in this context, it is a localized data structure within its own unique thread. A node has its own local memory, including imported modules and libraries, as well as its own command stack.

Nodes are hierarchical, and thus keep a connection to their parent nodes. Of course, this means nodes may also spawn child nodes, again each running on an individual thread. The lifecycle of a child is bound to its parent. When a parent node is terminated, all children first terminated. Any grandchildren and so-forth are terminated as well. Shutdown of a node hierarchy starts at the lowest canonical level of the family tree and gradually work upwards to the top-level parent node. Nodes that are currently busy may refuse to exit until they have completed their stack. If a node is not responding, or its work is not important, the target node may be force terminated, forcing all children to shut down as well.

**Memory Space**

Nodes contain their own local heap. This allows them to work independently of other nodes in a system without risking name or access collisions. However, the root node also initializes a global memory space which is accessible to all nodes. However, this memory space is protected with access safe guards. This makes working in the global space slower, but will allow common data to be accessed and modified anywhere in the program.

Sometimes a node may need to access information from another node. This can be done in three ways:

1. The requesting node may make a call to *request*, passing the handle or id number of the node who contains the desired information, as well as a string representing the symbol requested. This call is thread safe, and will not return object data until it is safe to do so. One important note to make is that is information is NOT linked to the original. It is a copy of the data. A call to *update* with the target node, symbol name and replacement info is required to update the symbol data. If direct access to the data is needed, a call to *request\_safe* may be called. This returns a thread-safe handle containing flag information, signaling both the availability status as well as permission level of the object.
2. The node owning the information may set the object in memory to a *public* visibility. In this instance, another node may use a handle pointing to the owner and dereference it as if the node were an ordinary structure. This will call a thread-safe getter or setter.
3. The easiest way is, of course, to simply shove the data in question to the global memory space. All references to the global space are thread safe. However, this means that if multiple nodes are using the same piece of information, it may be difficult to ensure that the memory isn’t corrupted or otherwise changed to an unexpected value. This can also cause issues with name collisions, especially if names within the global space occur elsewhere in the program. This approach is **NOT** recommended. While the global space is useful in certain instances, it should be used sparingly.

Types

There are several standard data types built directly into the engine. While they are not necessarily hard-coded into the logic, and indeed are treated much the same as any datatype that can be imported from a module, these are defined at startup have some slight speed advantages over imported types.

Listed below are the built-in data types and their range and limitations.

|  |  |  |
| --- | --- | --- |
| Name | Keyword | Information |
| integer | int | 16-bit signed integer |
| unsigned integer | uint | 16-bit unsigned integer (x64 only) |
| long integer | lint | 64-bit signed integer |
| unsigned long integer | ulint | 64-bit unsigned integer (x64 only) |
| character | char | 1 signed byte, text |
| byte | byte | 1 unsigned byte |
| string | string | Character array |
| array | arr | Generic dynamic array |
| function | func | Scoped, executable method with argument list |
| void | void | No data |
| class | class | Collection of members and functions |

Keywords

Built into the language are a variety of keywords that provide basic control and utility. In the previous section, data types and their keywords were listed, and shall be omitted in this section.

Keywords may take zero or more arguments.

**IO**

**Print**

Arguments:

*generic*

About:

Print is the most basic output call. Reading through an argument list, each value is converted into a string and printed to the screen. Escaped characters are processed normally. In the event that a non-primitive object is presented, the conversion table in the **Data Manager** is first checked for an appropriate string conversion. If this is unavailable, the name of the object type is printed instead.

**Get**

Arguments:

*none*

About:

Get is the most basic input call. This command reads a single line from the terminal (even if the node is not the assigned owner of the input handle, so make sure that the node id to the left of the terminal indicator is correct).

**Modules and Libraries**

**Load:**

Arguments:

string list

About:

The *load* function is used for loading and executing scripts. These may be bcx or knx files. When these scripts are loaded, they are stored in bytecode form as *pages*. In this regard, the scripts may also be unloaded to conserve memory when not in use. Additionally, the *load* method may be used with the ~t option to unload immediately after use.

**Import:**

Arguments:

string list

About:

This function is context dependent; depending on the extension of the file being imported, a different procedure will take place to load it into memory. The important thing to note here is that importing a script, bytecode file or binary module does not execute said resource. Instead, the RTE will attempt to store the information to be accessed later on, in contrast to the *load* keyword which will execute immediately, and only works with script extensions.

Bytecode

Scripts

Operators

There are a variety of operators available for performing arithmetic, logical and comparison operations. Operators are between one and three characters long and may or may not have an operand on the left-hand side.

Below are a few lists by category of operators and their functions.

Math

|  |  |
| --- | --- |
| + | Returns the sum of the left and right operands |
| - | Returns the difference of the left and right operands |
| \* | Returns the product of the left and right operands |
| / | Returns the quotient of the left and right operands |
| % | Returns the remainder of the left and right operands |
| ^ | Returns the left operand to the power of the right |
| √ | Returns the root of the right by the factor of the left. If the left operand is not given, it is assumed to be two. |

Logical

|  |  |
| --- | --- |
| & | AND Return 1 if left and right are non 0 |
| | | OR Return 1 if either operand is non 0 |
| ! | NOT Flip the right-hand operand to 0 if non-zero, or to 0 if 1 |
| !& | NAND Return not-and |
| !| | NOR Return not-or |
| |! | XNOR Return exclusive not-or |
| || | XOR Return exclusive or |
| ^& | Bitwise AND |
| ^! | Bitwise NOT |
| ^| | Bitwise OR |

Comparative

|  |  |
| --- | --- |
| == | Returns 1 if both operands are equal. Values, not addresses, are checked for equality |
| != | Returns 1 if operands are non-equal. Values, not addresses, are checked for equality |
| > | Left greater than right |
| >= | Left greater than or equal to right |
| < | Left less than right |
| <= | Left less than or equal to right |
| <> | Left is not between next two operands (comma separated) |
| >< | Left is between next two operands (comma separated) |
| <=> | Left is equal to or outside next to operands (comma separated) |
| >=< | Left is within or equal to next to operands (comma separated) |

Setters

|  |  |
| --- | --- |
| = | Set left equal to right |
| += | Set left to sum of both operands |
| -= | Set left to difference of both operands |
| \*- | Set left to product of both operands |
| /= | Set left to quotient of both operands |
| =& | Set left to bitwise AND of both operands |
| =| | Set left to bitwise OR of both operands |
| =! | Set left to bitwise NOT of right |
| ?= | Ternary operation |

Grouping

|  |  |
| --- | --- |
| ( | Begin priority or argument group |
| ) | End priority or argument group |
| { | Begin code block |
| } | End code block |
| [ | Start explicit list block |
| ] | End explicit list block |

Misc

|  |  |
| --- | --- |
| ~ | Option string |
| : | Index |
| . | Member dereference |
| # | Single line comment start/end |
| #\* \*# | Multi-line comment |
| , | List seperator |
| @ | Return address of following object |

Option Strings

Options strings provide a means of adding additional, optional control to a method or declaration without requiring long, cumbersome optional parameters. Option strings do not have enforced meanings, especially in third party implementations.

An option string is used by adding a tilde ‘~’ to the right of a method’s argument list (or simply to the right of the keyword if no arguments are given) followed by a series of case sensitive letters. Numbers may be used, but are discouraged. Options do not apply to variables.

Below are the suggested meanings of some option characters, and the most likely meaning throughout the standard implementation.

|  |  |
| --- | --- |
| f | Force method to continue through errors |
| w | Suppress warnings from this method |
| e | Treat warnings as errors |
| v | Print extended outputs |
| d | Enable debugging for function |
| g | Preference for global memory space |
| l | Preference for local memory space |

Libraries/Modules

Libraries allow one to easily access a swathe of functions, classes and other useful pre-programmed assets that would otherwise need to be written from scratch each project. However, many libraries can become bloated and slow, especially when written for as general a purpose as possible in the native language. Thusly, there are three possible ways to use libraries and modules in KNX, based on the degree of performance vs. convenience required.

The first is of course, script libraries; these are native knx scripts that must be loaded, tokenized, and then executed. While this is useful in the sense that scripts may likely be written much easier than binary plugins, as well as having the benefit of working even when made using an outdated version of the KDK, these are significantly slower than binary plugins as they must first be interpreted. This may, however, be slightly mitigated using the *import* keyword as one would with a binary plugin. When a knx or bcx file extension is imported, the RTE converts these to bytecode pages and caches them, allowing for rapid use. In this event, the scripts are NOT read upon load, but rather stored for future use.

The second option uses the bcx file format just mentioned. BCX stands for Byte Code X. Scripts may be converted from knx scripts to bcx using the *optimize* command, and passing in the path to a knx script. If no destination is specified, the knx script will be converted and the appropriate bcx will be placed in the origin directory. These bytecode files can be trivially scanned, and all dependencies required to use them are listed. The downside is, these are no longer editable by humans after being converted to a binary format.

Lastly, the third option; binary modules. By using the KDK, c dynamic libraries may be created in a way that easily interfaces with the RTE for implementing custom high-speed functions that may be able to perform actions that the vanilla engine is not designed to do. This allows immense customization for jobs where other libraries may be lacking or unable to perform the work needed.

Command Line Options

Command line options are passed into the program when it is invoked. POSIX style options are utilized. Options may be single characters, strings of single character options, or full strings that represent a single option, often with the potential to accept an argument.

Single dashes represent single character options. There may be one or more characters following a single dash, with each character specifying a different option. When an option starts with two dashes, the string following is considered to be a single option. These options may use the ‘=’ operator if supported in order to pass an argument to the option. All options and arguments must be done without spaced. If spaces are needed, be sure to wrap the string in quotes.

Below are the supported CMD options.

|  |  |
| --- | --- |
| -v | Set output to verbose |
| -V | Print version info for all components |
| -d | Enable debug mode |
| -h | Print help dialogue |
| -e | Treat warnings as errors |
| --maxmem={int} | Set max allowed memory in megabytes |
| --maxthread={int} | Set max allowed threads. Cannot be less than maximum nodes |
| --maxniodes={int} | Set max allowed nodes. Cannot be greater than maximum threads |
|  |  |

Versioning

Versions are set by hand before deployment, at least for now. For the sake of ensuring compatibility (or at least when diagnosing this as an issue), each component built with the KDK holds a both a version number consisting of a major, minimum and patch number, as well as an incremental build number that is only updated by the KDK, but allows each component to reflect which version of the KDK was used to create the component or module. This is mostly important for binary plugins, and when updated only certain components by hand. The version number is primarily for specific reference to a component; the most important number is the build number. If the build number is the same across all components, the system is probably stable.