KNX Runtime Language

2016

V 1.0.0 A

**Abstract**

This runtime language platform is a tool designed to make advantage of multiprocessing for applications such as scientific modeling and for automating tasks.

In general, this language is designed with high-control applications in mind. That is, this engine is designed for the creation and application of medium to large autonomous systems that facilitates a human operator.

Other viable applications may be in quick-and-dirty graphical applications and as a simple multi-purpose language.

Features include Object Orientated Programming (OOP), a multithreaded node based design as well as a module construction to allow for third party expansion.

The following sections of this document will explain further into the workings of the engine, as well as providing examples and tutorials of some of the more fundamental points of using this tool. For further reference on specific functionality, please refer to the API guide.

*Table of Contents*

|  |  |
| --- | --- |
| Description | Pg# |
| Command Line Options and System Configurations | 1 |
| Architecture | 2 |
| Node Hierarchy, Communication and Permissions | 4 |
| Data Types |  |
| Memory Structure |  |
| Object Orientation and Inheritance |  |
| Operators and Logic |  |
| Keywords |  |
| Flags |  |
| Terminal Features |  |
| Optimizations |  |
| Thread Synchronicity and Safety |  |
| General Syntax |  |
| Program Design |  |
| Versioning |  |
| Platform Dependencies |  |
| Compilation from Source |  |
| KNX SDK |  |
| KNX RTE |  |
| Building Custom Libraries |  |
| Debugger |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Tutorials |  |

**Command Line Options and System Configurations**

Arguments passed in through the command line allow for session configurations, as well as certain pre-compilation functions such as output of version information and project creation.

All command line options are expected to be preceded by a single or double dash (‘-’ or ‘--’) with no white space in-between. A single dash assumes that the following option will consist of a single character. If multiple characters follow a single dash, each will interpreted as an individual single-character option. Double dashed options may only specify one option at a time, and are expected to have one or more characters.

A non-dashed string or series of strings is considered an argument to be passed to the last stated option.

For example, the option “-vb –-src test” consists of three options: version, build and src (which targets project “test”).

The current command line options and their expected arguments are listed below. Arguments preceded by ‘\*’ are optional. Left out optional parameters revert to a default.

|  |  |  |
| --- | --- | --- |
| v | none | Output current interpret version |
| h | none | Output command line help printout |
| w | \*Warning... | Suppresses given warnings. If no warnings are specified, all warnings are disabled. |
| e | \*warning... | Considers the given error codes to be fatal, and thus end execution. When no parameters are given, all warning and errors will terminated execution on the current node. |
| d | \*verbose | Enable debug mode. If “verbose” is specified, output information will be much more detailed than normal. |
| m | File.knx,... | Accepts a file list list and produces a miniaturized version named *file.min.knx* |
| maxmem | Size (kb) | Maximum memory allowed by the engine. |
| maxnode | Size | Maximum number of nodes allowed to run at once. |
| maxthread | Size | Maximum number of threads to run on. By default the interpreter will run on all threads. |
| log | \*Location | Write events to a log file. If no location is specified, the log will be written to Logs/log[*index*].txt where *index* is the next log number. |

**Architecture**

The principle design consideration in KNX is the *node-based* processing scheme. In this, multi-threading is taken advantage of, with each node taking individual responsibility for processing data and managing resources for its particular scope.

Each node is part of a hierarchical tree. The root node of this tree is initialized as soon as the interpreter is run. Each new node spawned becomes a child of the node that spawned it. Parent nodes hold command and data authority over their children, allowing propagation of signals and scoped data access, while also constraining child persistence to the life cycle of the parent.

**Node Hierarchy, Communication and Permissions**

**Standard Input**

At the instantiation of the root node, access to the standard output is registered and restricted. Only one node at a time may hold the rights to the standard input, which is the root node, by default. However, other nodes may request standard input access, either temporarily or permanently, although only one node may hold standard IO access at one time.

**Node Access Permissions**

Like variable instances, access permissions extend also to nodes themselves. However, unlike most variables, the visibility permissions affect the information held in a nodal memory space, not the node itself; similar to class memory.

Nodes are, by default, protected by a *legacy* permission restriction. However, while the restriction keywords are the same for both variables and nodes, the meanings are different for the different contexts.

The table below details the effects of the permission keywords on nodes.

|  |  |
| --- | --- |
| public | Accessible to all other nodes. |
| private | Accessible to only the root node. |
| protected | Accessible to only the current node’s family branch. |
| legacy | Accessible to only parent nodes. |

**Data Types**

|  |  |  |
| --- | --- | --- |
| Type | Keyword | Description |
| integer | int | 32-bit integer (–2,147,483,648 to 2,147,483,647).  Default value:: 0 |
| Unsigned integer | uint | 32-bit unsigned integer (0 to 4,294,967,295)  Default value:: 0 |
| 64-bit integer | long | 64-bit integer(–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) |
| 64-bit unsigned integer | ulong | (0 to 18,446,744,073,709,551,615) |
| real | real | 1.7E +/- 308 (15 digits) |
| character | char | –128 to 127. Compatible with string manipulation. |
| Unsigned character | uchar | 0 to 255. Compatible with string manipulation. |
| Wide character | wchar | -32768 to 32767. Allows unicode. |
| string | string | An array of signed characters. |
| wide string | wstring | An array of wide characters. |
| bit-field | bit[n] | A bit array consisting of *n* bytes. Bits are accessed by by the [] operator. |
| class | class | Wraps variables and functions within an object. Classes may be inherited, and have a set schema upon their creation. These may not be modified individually, although it is possible to update the class definition which in turn will take effect across all current instances. |
| dynamic class | dynamic | Similar to classes, with the difference that dynamic classes do not have a defined schema and cannot be inherited. However, any member may be defined, gotten and set directly by referencing a member. If a member that does not exist is set, it will be created. If a non-existing member is gotten, it will return null. |
| function | func<type> | Creates a function object that may be invoked |
| array | array<type>[n] | Creates an array of type *type*. Size *n* is optional. If *n* is not declared, then the array initializes at size 0. If the array is set with a group list *{obj0, obj1,…}*, the size is set to that of the given list size. |
| collection | col[n] | Initializes an aggregate array of size *n*. If *n* is not declared, then the array initializes at size 0. If the array is set with a group list *{obj0, obj1,…}*, the size is set to that of the given list size. |
| keypair | keypair<key type, value type> | Creates a searchable keypair list. |
| file | file | A file IO object capable of both read and write operations. |
| data stream | stream | A constant information IO stream. |
| node | node | Spawns a new child node. While this node is maintained internally, a handle is also generated in which communications between nodes may be referenced apart from the *call* command. |
| void | void | A generic type which contains a reference to the target memory. This acts as a conventional pointer, and can be used to modify, identify and destroy a target, and change targets when needed. |

**Memory Structure**

**Memory Storage**

All memory stored internally is initialized by pointers. However, garbage collection is not used in the current version of KNX. Instead, variables expire at the end of its scope, and are then released, unless otherwise specified.

**Scope**

Identifiers are resolved by the highest scope in which it has been declared. If a symbol has already been defined in a lower scope, the most recent is used instead.

Functions provide a temporary scope. At the end of a method, all existing objects declared within the method are destroyed, unless declared with the *persist* modifier.

**Permissions**

By default, all memory declared is publicly visible within the node namespace. However, additional restrictions exist between nodes, and other restrictions may be placed on a per object bases.

Listed below is a table of variable permissions and their associated keywords.

|  |  |
| --- | --- |
| public | Accessible everywhere from the objects scope and up. Is accessible to other nodes if the host node allows. |
| private | Accessible only to the current node. If declared within a class, the object is only visible within the class that defines it. This object is not inheritable. |
| protected | Accessible only to the current node. |
| legacy | Publicly accessible within the current node, but only accessible from parent nodes. |

**Object Orientation and Inheritance**

**Classes**

Classes are data wrappers that encapsulate a set of members and functions within each instances’ scope.

There are two types of classes: *class*, and *dynamic*.

The ordinary class consists of a rigid schema determined upon definition. Any memory or methods declared within a class may utilize the four permission modifiers, which are *public* by default.

Dynamic classes are much more flexible, but also far more limited. While a dynamic class may hold memory and functions(in the form of references or lambda expressions) members may only be of public visibility. This type of class may not be extended or inherited.

The most obvious difference in behavior is that, while standard classes are defined with particular schema, dynamic classes have functions and variables added or removed at any time, and therefore may have unpredictable members.

Standard classes are defined with the *class* keyword, as well as a member list with each members’ initial value (if no initial value is specified, the default value will be used instead), as in the case below:

|  |
| --- |
| *#example\_class.knx*  class demo{  real v1, v2;  int v3;  #a class defined function  int product(real a, real b){return a\*b;}  #pure virtual function  virtual void lambda();  } |

Dynamic classes are defined differently.

|  |
| --- |
| *#example\_dynamic.knx*  dynamic demo;  demo.integer = 1.0;  demo.str = “The number is: “;  print(demo.str + demo.integer);  #output: The number is: 1 |

**Operators and Logic**

**Assignment Operators**

|  |  |  |
| --- | --- | --- |
| = | Set |  |
| != | Set Not |  |
| += | Add set |  |
| -= | Minus Set |  |
| \*= | Multiply Set |  |
| /= | Divide Set |  |
| %= | Modulus Set |  |

**Math Operators**

|  |  |  |
| --- | --- | --- |
| + | Plus |  |
| - | Subtract |  |
| \* | Multiply |  |
| / | Divide |  |
| ^ | Power |  |
| % | Modulus |  |
| √ | Root |  |

**Access Operators**

|  |  |  |
| --- | --- | --- |
| . | Member access |  |
| ?. | Safe member access |  |

**Logical Operators**

|  |  |  |
| --- | --- | --- |
| & | And |  |
| | | Or |  |
| ! | Not |  |
| !& | Nand |  |
| !| | Nor |  |
| || | XOr |  |
| |! | XNor |  |

**Comparison Operators**

|  |  |  |
| --- | --- | --- |
| == | Is equal to |  |
| != | Is not equal to |  |
| > | Greater than |  |
| < | Less than |  |
| <= | Less than or equal to |  |
| >= | Greater than or equal to |  |
| L<=>R | Outside L and R, inclusive |  |
| L>=<R | Within L and R, inclusive |  |
| L<>R | Outside L and R, exclusive |  |
| L><R | Within L and R, exclusive |  |

**Modification Operators**

|  |  |  |
| --- | --- | --- |
| @ | Code-level | A statement or flag following this operator will assumed literal by the interpreter, and additional metadata will be passed along without consumption. |

**Miscellaneous Operators**

|  |  |
| --- | --- |
| # | Comment |
| #\* | Multi line comment start |
| \*# | Multi line comment end |
| ##\* | Documentation comment start. This will only be read by a call to **makedoc**, providing the commented file in question. |
| \*## | Documentation comment end |
| <identifier, value> | (Within documentation comment only) Creates a document line |

**Keywords**

Keywords provide build-in functionality, such as variable declaration, loops and other processing methods. The syntax of keywords can behave the same as functions in the sense that function parameters may be given to them. However, if no argument list is provided, each keyword will accept the next token provided as an argument.

**Data**

|  |  |
| --- | --- |
| Int(“name”, value) | Creates an integer |
| uint(“name, value) | Creates an unsigned integer |
| long(“name, value) | Creates a long integer |
| ulong(“name, value) | Creates an unsigned long integer |
| real(“name, value) | Creates a real number variable |
| char(“name, ‘value’) | Creates a character |
| uchar(“name”, ‘value’) | Creates an unsigned character |
| wchar(“name”, “value”) | Creates a wide (16-bit) character |
| string(“name”, “value”) | Creates a string |
| wstring(“name”, value) | Creates a string of wide characters |
| bit(“name”, size) | Creates a bitfield of *size* bytes |
| class(“name”, {value}) | Creates a standard class |
| dynamic(“name”) | Creates a dynamic class |
| array(“name”, type, value….) | Creates an explicit type array |
| collection(“name”, value..) | Creates a general type array |
| keypair(“name”, keytype) | Creates a hashed keypair table |
| file(“name”) | Creates a file stream handle |
| stream(“stream”, type) | Creates an IO stream handle |
| node(“name”) | Creates a child node and returns a handle to that node |
| void(“name”) | Creates a void type |
| object | Creates an object, the basic element all variables are derived from |

**Control**

|  |  |
| --- | --- |
| if(boolean) | Basic conditional control |
| else | Executes if preceding *if* controls fail |
| for(var, condition, end operation) | Provides a conditional loop |
| while(condition) | Provides a simplified conditional loop |
| switch(value) | Executes an expression by case |
| case value: expression | Assign an execution point in a switch |
| break | Exit a loop or switch body |
| throw | Throws an exception |
| catch(exception) | Catches an exception (if no argument given, exception is consumed) |

**Modifiers**

|  |  |
| --- | --- |
| public | Will set a variable to public visibility within its scope |
| private | Will set a variable to private visibility within its scope |
| protected | Will set a variable to protected visibility within its scope |
| legacy | Will set a variable to legacy visibility within its scope |
| static | Prevents a variable from destruction upon exit of scope. Only one instance of a particular variable declared with static may exist at one time. Initialization will be ignored if an instance of the variable already exists. This is the same as using the ~s flag. |
| final | Prevents standard classes from being extended, functions from being modified, or variables/functions from being overridden. This is the same as using the ~F flag. |
| global | Declares a variable within the root node. This is the same as using the ~g flag. |
| aligned | Assignable only to standard classes during declaration. This modifier plans out the memory mapping provided, instead of fitting memory in the format provided. This may be invoked by the ~o flag. |

**Optimization**

|  |  |
| --- | --- |
| cache (“name”, {...}) | Cache may be given a function body. This code will not execute at the time *cache* is invoked, but is instead tokenized and stored by the given name, which may be invoked as a normal function.  Caching removes the need for the interpreter to tokenize code each time a function is called, drastically increasing performance. However, once a function is cached, it becomes immutable.  If *cache* is called when an object of the same name already exists, it will be ignored. |
| shrink(function) | The *shrink* keyword miniaturizes a given function by formatting its plain-text representation. This optimization is not as fast as *cache*, however, but does preserve mutability. |

**General**

|  |  |
| --- | --- |
| import | Import a module |
| include | Import a \*.knx file |
| module | Creates a namespace in which variables, values and methods may be declared without conflicting with similarly named entities outside of the namespace. |
| this | Begins the resolve chain at the current scope. |
| makedoc | Scans a given file (or list of files) and generates documentation files for each, if documentation comments are found. |

**I/O**

|  |  |
| --- | --- |
| readline | If the calling node has input access, the current terminal input buffer will return the next full line supplied by the user. |
| readchar | If the calling node has input access, this function will return the next keystroke supplied by the user. |
| printline | If the current node has output access, the terminal will print a newline (\n on unix, or \r\n on windows) to the terminal following the supplied argument. |
| print | If the current node has output access, the terminal will print the supplied argument. |
| request\_i | Requests exclusive standard input permission. Returns -1 if failed. |
| release\_i | Releases exclusive standard input permission. |
| request\_o | Requests exclusive standard output permission. Returns -1 if failed. |
| release\_o | Releases exclusive standard output permission. |

**Flags**

Flags are modifiers that may effect everything from declarations to visibility protections to simply hint at how a variable or function should be interpreted.

A flag is stated by using the *flag operator (~)*, followed by a single (or multiple) case-sensitive characters. If multiple characters follow a single ~ operator, they will all share a target. Again, these characters ARE case-sensitive, so the two flags *~f* and *~F* flags can mean two very different things.

Depending on the context, flags may be consumed by the interpreter, or passed along with an argument to be deciphered by the called function. In the former case, if a flag were to be consumed, it shall be removed from the passed flag listing for the target. In this event, if a flag had meaning to the interpreter that was intended to be passed along to another entity, said entity would not receive it. In this event, the *code-level* operator (@) may precede the individual flag to prevent the interpreter from consuming the flag. If the code-level operator precedes the full flag declaration (*ie. “@~fg”*), then the entirety of the flag set will not be consumed by the interpreter.

If a flag is not consumed, and is passed into a function, the *switch* statement may be used to analyze the input. When analyzing flags, the switch statement will execute all flags on a target simultaneously, instead of requiring a loop.

|  |
| --- |
| flag\_example.knx  void nit\_vars(int val){  switch(val.flags){  case ‘g’: int integer\_variable = val ~g; break; #manual handling of standard flag  case ‘x’: print(“x flag was specified in init\_vars function”); break; #handles function specific flag, not normally consumed  }  } |

**Thread Synchronicity and Safety**

KNX is designed around multi-threading. The core language of KNX is thread-safe. This section is designed for module developers using KNX SDK to develop extensions for the core language.

**Program Design**

In program design, there is no one best design practice. However, in KNX does promote some certain design practices. In particular, use of optimizations, functions and classes. While KNX does not enforce style, in general a standardized styling improves readability across projects made by separate developers.

**Third Party Scripts**

When developing third party scripts, or even to encapsulate a particular private project for more organization and collision safety, using the *module* wrapper is advised. Not only does this allow module-scope variables to be declared without worry about conflict or corruption from other scripts, but also helps package each library or program into a manageable bundle.

**Miniaturization**

Like popular scripting languages such as javascript, KNX may be miniaturized to decrease load time. This may be done with the *-m* command line argument, which accepts a list of files to miniaturize. The new smaller files are named the same as the original, but with *.min* between the file name and extension, as in; *example.min.knx*.

**Versioning**

**Platform Dependencies**

**Compilation from Source**

**KNX SDK**

**Building Custom Libraries**

The KNX runtime is built with a separate dependency. The KNX Development Kit, or KDK. Opposed to KNX written libraries, KDK modules provide a much more flexible and quick library written in C which may be dynamically loaded via the *import* keyword.

**SDK API**

The KNX SDK API provides a number of interfaces useful in providing easy, uniform communication between the core runtime environment and imported modules. The following section will be focused on targeting custom module development for the KNX platform.

**Module Development**

Modules are developed in the C language and compiled as dynamic libraries. Primary, SDK interfaces allow for the development of new binary functions, object, and module managed services.

Classes and functions defined by a module may be defined as objects and registered with the core program’s data registry. In this way, without the interpreter directly being able to know the details of a module’s implementation, psuedo-lambda expressions included in the definitions of the objects may be used to customize these functionalities as efficiently as possible, while still allowing the flexibility of a modular system.

Module implemented classes are directly designed in the fashion that a KNX class is stored during runtime interpretation. However, modular systems have certain differences.

* In first, modular class schemas are completely immutable. As such, libraries may retain a higher degree of reliability in a project-by-project basis.

**API**

|  |  |  |
| --- | --- | --- |
| **method** | **arguments** | **description** |
| registerModule | char \* namespace,  Version version | This must be called before registering any classes or methods. This information allows the core interpreter to manage this libraries resources. |
| registerClass | char \* name,  Memscheme scheme,  {obj,…} | Registers a class to be transferred to the core engine upon this library being loaded. Accepts a name, a memory scheme and a list of objects to populate the class with. If Memscheme is null, the core engine will attempt to determine the best scheme to use automatically. |
| registerMethod | char \* name,  char \* returnTypeName,  Type \* argumentTypeList,  Lambda (returntype, (objects \* objList...)){expression} | Registers a method within the module domain. Accepts a name (which will then be registered as a type), an expected return type and a list of expected argument types. This is only required for publicly visible methods. |
|  |  |  |

**Definitions**

Because the SDK is responsible for providing a common framework for both the runtime environment and additional plugins or utilities, most of the data structures and many method stubs used in the RTE are defined in the SDK. This includes definitions for nodes

**KNX RTE**

**Objects**

The most primitive object type is simply defined as the *object* structure. This provides storage, registration and data management information about memory extended from this structure. This is required for all variables and functions to be understood by the KNX runtime engine.

**Classes**

KNX classes consist of an identifier and a some sort of list of member objects. These must be registed with the core engine with the use of the *registerClass* function, in order to be recognized by the core runtime engine.

Once registered, a class will become a recognized type accessible through KNX script.

**Functions**

Module defined functions must follow a standard API designed to allow the registration and passing of data to a method not recognized by the core engine. This can be somewhat tricky to do. However, it is possible through the use of anonymous functions.

**Memory Management**

There are several ways in which KNX attempts to manage memory in the most efficient way possible. Based on the type of class or function being declared in the package, a memory management system may be declared to specify the best way to package and maintain memory for faster access during runtime.

The enumeration *Memscheme* offers a selection of managements procedures that may be used to manage object member organization.

The table below lists the options and their effects.

|  |  |
| --- | --- |
| Memscheme.list | Uses a standard linked list to lookup members. No active management occurs. This is best for small classes. |
| Memscheme.binary | Utilizes a binary tree based on the name hashes of the given objects. No active management occurs. |
| Memscheme.splay | Utilizes a binary splay tree. Each time an element is accessed, it is splayed to the top, effecting the position of other leaves along the way. |
| Memscheme.repeat | Similar to the list, each time a member is selected, it is moved to the front of the list. This is useful if certain members are expected to be accessed repeatedly. |