# X10GIMLI

**X-10 General Interface Modal Language Idea**


# Documentation and Reference Guide

Language Concept and Design by
Adam Lane

# TABLE OF CONTENTS

# Introduction

## X10GIMLI Background

The **X-10 General Interface Modal Language Idea** (X10Gimli) came about while searching for a good software package for controlling home automation devices, principally X-10 devices. The systems I had seen did not provide the functionality I desired for controlling home automation devices. I had several specific wishes for a home automation system. It would need to:

1. Allow multiple control modes for a single house code so that one remote control can command different types of things depending on the current control mode.
2. Enhance the functionality and configurability of motion detectors by intercepting their commands before sending them to devices.
3. Toggle controls to respond to or ignore commands.
4. Add computer and internet functionality as responses to X10 commands.
5. Allow triggered responses based on different kinds of timed events and X10 events.
6. Handle simple macro definition for complex responses to system events.
7. Implement different system modes that have different event triggers, control modes, and operating conditions.

With those goals in mind, I designed a simple language that could handle the needs that I initially specified. The preliminary system worked fairly well, but I soon discovered that there were other features that I desired, and I began to expand the capabilities of the language to handle:

1. X10Gimli specific data types and arithmetic rules and operations between them.
2. Standard packet style used for all data coming into the system.
3. User defined functions that can return values, instead of macros.
4. Conventions for utilizing user created Java classes in the language.
5. Support for common and necessary language constructs, such as if/else, loops, variable definitions.
6. File import capabilities for system extensions and modularized structuring within modes.
7. Distribution of system components and interfaces over a network.

At this point the language has evolved to have the necessary functionality to handle all of my home automation wants and needs. A set of design conventions allows for new input gateways, external function calls, and external variables to be created without much difficulty. By following the conventions defined for adding new language functionality, external functions, variables, and gateways become automatically available within the language. So, the language itself has all of the functionality that I need, though new external functions and gateways still need to be created and always will.

I still have desire for more functionality within the language, but most of it is syntactic sugar that can already be achieved with the available syntax. The most important functionality is already there, so, I've determined that X10Gimli may be useful for others in my plight, and I'm preparing it for distribution.

# Getting Started

Before anyone can take full advantage of the X10Gimli system, som minimal qualifications can be considered. Information regarding actually using X10Gimli can be found in the **System Usage** portion of this document.

## Who Needs X10Gimli?

X10Gimli is intended to allow the technical complexity and ease of implementation that programming languages allow. With that in mind, technically minded individuals probably stand to benefit the most from a home automation programming language. At any rate, people who could use X10Gimli include:

- People who are not satisfied with the many home automation products.
- Those who are seeking a cross platform solution to home automation.
- Individuals interesting in a fully distributed home automation system.
- Anyone who is not afraid to try new ideas that can be easily integrated into a large system.
- Persons with a

Certainly, there are many other reasons to use X10Gimli, but all are related to the expansive control that is possible with this system.

## What is Needed to Run X10Gimli?

The X10Gimli language interpreter is written completely in Java and none of the inner workings of the system require any native libraries or features. With that, any system with a valid Java Runtime Environment should be able to make use of X10Gimli. That does not mean that all existing external gateways, functions, and identifiers will allow the same cross platform functionality.

Many externally created functions, gateways, and identifiers will be implemented with hardware and software dependancies that vary between platforms. This is unavoidable. However, the level of abstraction provided to implement such interfaces allows the platform dependant portions to ported with a minimal amount of other modifications.

Of cource the actual X10Gimli software is required before it can be used. At present, the distribution of the X10Gimli system is undetermined.

So, simply put, to minimally run X10Gimli, one only needs:

- A computer with a valid JRE installed.
- The X10Gimli distribution JAR file.

That minimal setup would allow X10Gimli to be launched, networked, and serve as the hub of a distributed home automation system.

For enhanced functionality, individual class files that act as external functions, identifiers, and gateways are required. Anyone can develop new class files for these purposes, and distribute them. Each user defined system component may require that certain packages be already installed, or may require of native components before becoming fully functional. Such requirements should be described with any new gateway, function, or real-time identifier.

# System Usage

## System Configuration

### X10GIMLI Properties

The X10Gimli system uses a set of properties from a `.X10GIMLI` file to prepare the system to load, log, and execute an X10Gimli program. The properties file contains information regarding the following system parameters:

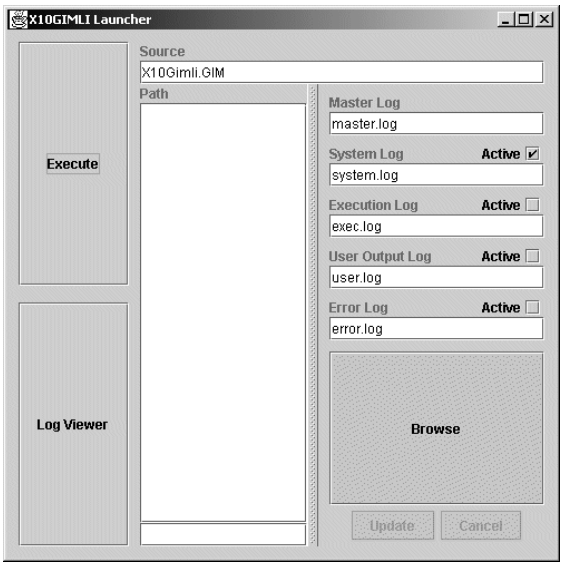
- primary source file to execute
- search path to load files
- log files to use
- which logs are active

These attributes can be configured individually in the properties file using any text editor. The properties file is the same as any Java properties files. Here is a sample file with all of the available attributes.

```
#X10GIMLI startup properties
#Mon Aug 27 11:10:14 MDT 2001
MASTER_LOG_FILE=master.log
NUM_PATHS=1
PATH1=..\\samples\\mediacontrol
SYSTEM_LOG_FILE=system.log
EXECUTION_LOG_FILE=exec.log
EXECUTION_LOG=FALSE
USER_LOG_FILE=user.log
ERROR_LOG_FILE=error.log
ERROR_LOG=TRUE
SYSTEM_LOG=TRUE
USER_LOG=TRUE
SOURCE=..\\samples\\mediacontrol\\MediaControl.GIM
```

All programs that are executed in X10Gimli require an associated properties file. Properties files can be editted manually using any text editor. Each log file has a file name associated with it, and another value that determines whether or not the log is active. The individual search path entries are numbered properties, and the number of search paths is also contained in the file. The most important property for executing any program is the source file, which is also specified in the properties file.

Although the properties can be editted manually, a GUI properties editor is also available.




The properties editor allows all of the load, log, and execution properties to be configured quickly and easily. It also insures the properties file format is correct and that no properties are missing from the file. It is the preferred way to handle configuration of the system before launching it. After the properties have been set, the "Exectue" button causes the system to run, using the properties specified. The log viewer can also be started before beginning execution. Accessing the GUI launcher is covered in the **Running X10Gimli** section.

## Paths

X10Gimli uses a list of search paths to handle loading source files. The search paths are used to find the primary source file, and all imported files. The paths can be specified with or without a terminating directory symbol. When attempting to resolve file locations using the search path, file names are appended to search paths and checked to see if files with such names exist.

The order the paths are search is the order that they fall in the properties list. The first path that successfully allows a filename to be resolved is the path that will be used to load the file. If there is more than one valid file, only the first file found will be loaded.

Files can be specified with absolutes paths that do not require path resolution. However, that requires hard coded absolute paths in individual source files, which will force certain directory structures to any program. It is preferred to specifiy file names and have them resolved using the specified available search paths. The current active directory does not need to be specified. It will always be searched before any specified paths. Also, relative paths can be specified an will be resolved relative to all search paths.

## Log Files

X10Gimli provides four unique log files, and a master log that are maintained during system execution.

- **System Log** - This log receives all of the information regarding the state of the system model. For example, all input received, all connections, interface initialization, thread intialization, mode changes, and such will be recorded in this log. In general this log should be enabled at all times. It does not incur much overhead, but it does allow a substantial amount of information to be tracked, and potentially backtracked under special circumstances.

- **User Log** - The `log` command provided with the language allows user entries to be specifically added to the user log. In this way, users can have specific information logged whenever user code is being executed. For example, one might want to log the number of times a particular motion detector was triggered during the day. Basically anything can be logged at the user's discretion.

- **Execution Log** - This is the most comprehensize of all the log files. It logs the execution of every single X10Gimli command. With this log, execution can be traced to individual function calls and command statements to determine code problems and determine points of failure. Generally, this log is intended for debugging purposes. It adds a great deal of overhead to the execution of the system, and can have a noticable effect on performance.

- **Error Log** - Since X10Gimli handles errors on the fly and attempts to recover from them. Many errors could occur that are not noticed. The error log records all illegal operations, interface failures, and other problems that can occur during system execution. In conjuntion with the other logs, errors and their causes can be backtracked.

- **Master Log** - All log active log output will always be combined into the master log as well as the individual logs. The master log allows all of the different log events to be viewed in the context of the other types of log events. Only the active logs will appear in the master log.

Each log can be individually enabled or disabled. Some performance enhancement can be gained by disabling logs, particularly the execution log. So all of these logs can be specified in the system configuration file, or configured in the GUI launcher.

## Source File

The program that X10Gimli executes is specified in the properties file. The source file is resolved using the provided paths, as described earlier. The source file becomes the primary system model that is used during execution. All imported files must occur starting in the source file.

When execution begins, the source file loads, along with all imports, and the startup commands for all files is executed. At that point, the system is ready for normal operation.

# Running X10Gimli

## Starting

There are a few ways to start the X10Gimli system, and execute a given properties file. All startup and execution is handled by the `X10GimliLaunch` program.

In Windows, with a valid Java Runtime Environment installed, double clicking on the X10Gimli.jar file will cause the GUI launcher to load using `default.X10GIMLI` as the properties file for editing and execution. This is the simplest method to begin executing a program. The drawback is that no different properties file can be specified in this way. So, input parameters must be passed to the launch program.

The launch program usage is given below:

**Usage:**

```
X10GimliLaunch [-g] [-l] [{filename}]
```

**Parameters:**

| | |
|---|---|
| *-g* | starts the GUI launch system |
| *-l* | starts and attaches the log viewer |
| *filename* | name of properties file to use |
| *-?* | provides this usage list |

If the GUI launcher is not started, the properties file will begin execution immediately. Note also that if no parameters are specified, then the default properties file will be loaded in the GUI launcher as previously described. Also, if the specified properties file does not exist, it will be created in the GUI properties editor.

Here are some examples of launching X10Gimli from the command line:

```
java –jar X10Gimli.jar –g x10gimlichat
java –cp X10Gimli.jar X10Gimli.X10GimliLaunch –l homecontrol
java –jar X10Gimli.jar controlprogram.X10GIMLI
java –cp X10Gimli.jar X10Gimli.X10GimliLaunch –l automate -g
```

As is shown, there are many ways to launch the system from the command line. Notice that the startup parameters can be specified in any order. Notice also that the properties file can be specified with or without the extention. If an invalid extension is specified, the `.X10GIMLI` extention will be appended. Remember that other Java parameters can be specified, and depending on the individual Java configuration, different classpath and run-time parameters may be desired or required.

To simplfy startups, the example command line parameters could easily be placed in a script, `.BAT` file, or Windows shortcut, to name a few.

When the system begins execution, the following startup sequence is followed:

- load startup properties
- initialize log files
- read source files
- start trigger check tread
- start model input handler thread
- run initialization commands for the system model

After that, the system should be running and ready to respond to input.

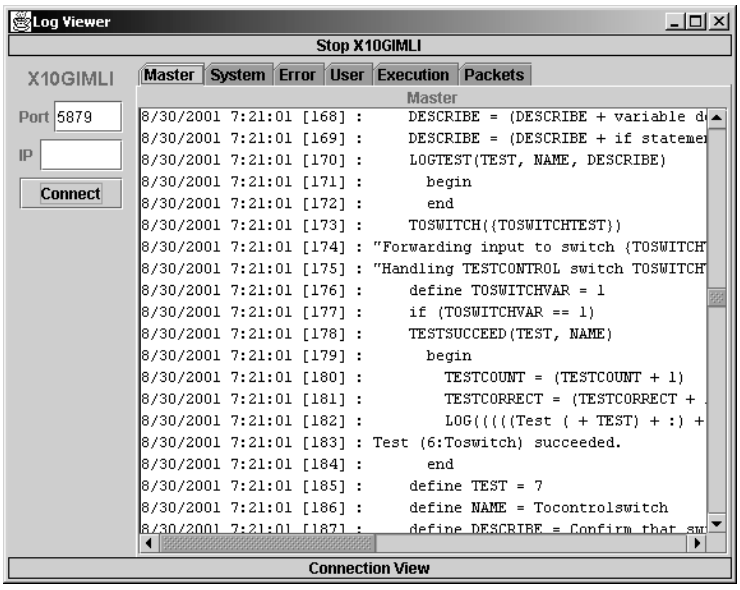
## Log Viewer

The log viewer allows log output to be viewed in real time. Every time a log event is output to the log files, the event is also packeted and sent out through the system model's output gateways. This allows log output to view viewed remotely by anything connected to the system's output gateways. The log viewer has a two operational modes:

- local
- remote

In 'local' mode, the log viewer is connected to the same `Debug` output class that the model is using to generate and send the log output. In other words, the log viewer is being run in the same virtual machine as the actual system model. This mode can not monitor all outgoing packets, because it is attached to the log generator, and not to the system model. So, only log packets will be received.

In 'remote' mode, the log viewer connects to the system model by way of the TCP gateway. The log viewer provides a connection panel to specify the connection port and address. When this type of connection is established, the log viewer receives all outgoing packets from the system model, and not only the log packets. This allows lower level debugging of the output connections. An image of a remote log viewer is shown:



As can be seen in the image, all of the different logs are viewable from the log viewer. The viewer also will display all incoming packets.

The log viewer is intended to provide the simplest implementation of a remote system monitoring application. It allows all system events to be monitored remotely, or locally and works well to debug X10Gimli programs. Future modifications to the log viewer will allow different log output to be enabled and disabled during execution, but this had not been implemented.

## Stopping

Since X10Gimli is intended to execute in the background and be invisible to users under normal circumstances, and since there is no direct way to control the system, shutting down the system must be done by a connection.

X10Gimli is signalled to shut down when it received a packet of type "X10GIMLI" with a COMMAND tag that contains "SHUTDOWN". When X10Gimli receives the command to shutdown, it immediately begins executing the system model's finish commands. After that, the master log is written to with the shutdown notification, and system will exit.

Any connected process can issue the shutdown command and from any location. However, the log viewer is the only thing provided that will send the shutdown command to the system model. The button at the top of the log viewer, labled "Stop X10GIMLI" will send the command to shutdown the execution model. To prevent an accidental shutdown of the system, a confirmation dialog will be issued: The shutdown must always before confirmed before actually sending the command.


After confirming the shutdown, the packet will be sent, and X10Gimli shutdown should proceed as described.

# Creating Source Files

## General Format

X10Gimli source files follow a format that is defined by the language grammar. Simplified, all X10Gimli files must fit the following template:

```
x10gimli programname;

imports
   "import1.gim";
   "import2.gim";

definitions
   def1 = 1;
   def2 = 2;

functions
   func1() begin end
   func2() begin end

triggers
   trigger1 (true) do begin end
   trigger2 (true) do begin end

start
   begin
   end

control <CONTROL1>
   [1] begin end
   [2] begin end

control <CONTROL2>
   [1] begin end
   [2] begin end

finish
   begin
   end

mode mode1
   imports
   definitions
   functions
   triggers
   start
      begin
      end
   control <CONTROL1>
   control <CONTROL2>
   finish
      begin
      end

mode mode2
   imports
   definitions
   functions
   triggers
```

```
   start
      begin
      end
   control <CONTROL1>
   control <CONTROL2>
   finish
      begin
      end

end x10gimli.
```

The source code template shows the order that all X10Gimli source files must follow to define the major system components. The bolded text represents keywords in the language, but not all keywords are shown.

When creating an X10Gimli program, any of the declaration sections can be left blank, or entirely ommited. The order that any declaration occur must still be followed, but things can be skipped. This means that the most simple valid X10Gimli program that can be written is as follows:

```
x10gimli program; end x10gimli.
```

## Specific Pieces

As was seen, there are several declaration sections that correspons to the major components of the system model and individual modes. The main system components are described as follows:

- **Imports** - Individual X10Gimli source programs and controls can be linked into any other execution environment by adding the file to the imports sections of the system model, or mode. Imported files act as an extension of the individual environments and allow imported functions, definitions, and controls to be available from within other source files.

- **Definitions** - Variables can be declared at system, and mode levels of visiblity at the beginning of a program. Definition can also be added during system execution, but knowing the visibility of such variables is more complicated when done later.

- **Functions** - Within any X10Gimli program, functions can be declared that use input parameters and can possibly return values. All user defined functions must be found in this section of source code. Individual modes can have functions declared that are only accessible within each mode, or override functionality of functions from the system model.

- **Triggers** - Permanent triggers can be defined in this portion of the source code. Triggers can be named or anonymous. Other triggers can also be defined in source code, but they are temporary and will only fire once. Triggers defined in this section are permanent and can fire an unlimited number of times. If the triggers are named, they can also have their functionality changed during run-time, but they remain permanent.

- **Start Command** - When beginning execution of an X10Gimli program, or when changing control modes, the start command for the model or mode is executed. This allows certain preparatory things to be done before entering a mode, or beginning the system. For example, all input and output gateways should be initialized in the start command of the system model.

- **Controls** - Controls are the most import part of the language because they allow responses to be defined for individual inputs. Each control should have a list of switches that define which individual inputs will be handled. Controls have an identification that defines what types of inputs they will respond to. Controls can be overridden between different modes, which allows unique modes to respond to the same input in different manners.

- **Finish Command** - Just as the start command is executed when beginning a program or entering a mode, the finish command is executed when exiting a mode, or shutting down the system. This allows anything to be resolved before changing modes. For example, state consistency can be maintained between modes using the start and finish commands.

- **Modes** - Individual modes can be defined that have their own variables, functions, imports, and controls. When a mode is active, the data from other modes is inaccessible. So, modes allow exclusive functionality in that way.

Note that although modes follow the same pattern as the system model, new modes cannot be defined within modes. More detailed information regarding the individual system pieces can be found in the **System Architecture** section of this document.

# Designing Simple Control Program

To assist users in getting started, this section will go through the process of creating a multi-mode "Hello, World" control program. This will demonstrate the important pieces of an X10Gimli program – different modes, controls, switches, functions, definitions, triggers, imports, and input and output gateways. Throughout this example, bolded text will represent the portions of the program that are added for each step of the creation progress.

Initially, the previously given template can be used to prepare the new program.

```
x10gimli HelloWorld;
end x10gimli.
```

Before we define any of our controls, we should define the modes we want to use. In this case, we will have two modes. One mode is for "Hello,world!", the other is for "Goodbye, world!":

```
x10gimli HelloWorld;

mode Hello
mode Goodbye

end x10gimli.
```

In any X10gimli program, the input controls need to be defined. For this case, we will use X10 control input. So, we add X10 controls for both modes:

```
x10gimli HelloWorld;
```

```
mode Hello
   control <X10>

mode Goodbye
   control <X10>

end x10gimli.
```

We want a function to handle doing the output for this program. Our output will be to the user log file, and to the text-to-speech gateway:

```
x10gimli HelloWorld;

functions
   output(text)
      begin
         log (text);
         speaktext(text);
      end

mode Hello
   control <X10>

mode Goodbye
   control <X10>

end x10gimli.
```

At this point, we can consider the input responses that we would like. For this, any ON command will be used to switch to Hello mode, and any OFF command will switchinto Goodbye mode. All other input commands will be used for doing output. We will do output using the output function we created:

```
x10gimli HelloWorld;

functions
   output(text)
      begin
         log(text);
         speaktext(text);
      end

mode Hello
   control <X10>
      [OFF]  changemode (Goodbye);
      [true] output("Hello, world!");

mode Goodbye
   control <X10>
      [ON] changemode(Hello);
      [true] output("Goodbye, world!");

end x10gimli.
```

For extra output, we will use a special trigger that fires every minute. To do this, we will also want a global variable that keeps track of the last minute to fire. That way, we can easily check the next minute. Every time our trigger fires, we'll give the current time:

```
x10gimli HelloWorld;

definitions
   lasttime = time;

functions
   output(text)
      begin
         log(text);
         speaktext(text);
      end

triggers
   minutetrigger (lasttime+1 == time) do
      output("Hello!  The time is "+time);

mode Hello
   control <X10>
      [OFF] changemode(Goodbye);
      [true] output("Hello, world!");

mode Goodbye
   control <X10>
      [ON] changemode(Hello);
      [true] output("Goodbye, world!");

end x10gimli.
```

And finally, we need to initialize the gateways that will be used in this program. We need an X10 input gateway and a text-to-speech gateway. The text-to-speechgateway also uses an imported file for the function `speaktext`. We also need to set the active mode at the start of the program. So, we add the finishing touches to the program.

```
x10gimli HelloWorld;

imports
   "TTS.GIM";

definitions
   lasttime = time;

functions
   output(text)
      begin
         log(text);
         speaktext(text);
      end

triggers
   minutetrigger (lasttime+1 == time) do
      output("Hello!  The time is "+time);
```

```
start
   begin
      initialize(MR26A, "COM1");
      initialize(TTS);
      changemode(Hello);
   end

mode Hello
   control <X10>
      [OFF] changemode(Goodbye);
      [true] output("Hello, world!");

mode Goodbye
   control <X10>
      [ON] changemode(Hello);
      [true] output("Goodbye, world!");

end x10gimli.
```

So, after all that, we have a program that will say "Hello, world!" and "Goodbye, world!" on most X10 inputs. It will change modes with ON and OFF commands. Also, it will say the current time every minute. This program demonstrates all of the significant components available in X10gimli source code. The remainder of the documentation explains in detail the functionality of the language constructs and contains more examples.

This particular example is given to show how a program can be created from scratch. A complex home control program should be given some thought before its implementation. To launch the example program, the instructions given in the **Running X10Gimli** section of this document can be followed.

# Language Overview

X10Gimli is an interpreted language designed to allow central handling of multiple input and output devices on a number of levels. The language is focused on home automation functionality. The language overview provides descriptive information regarding much of the language definition and internal functionality. The information provided is intended to offer an in depth look at the intricacies of the language, and to provide a standard that the language implementation should meet.

# Language Definition

The basis of the language functionality stems from the low-level definition of the language. Before the exact language semantics and features were defined, a general language grammar was developed. As the internal functionality and aspects of the system were built using the language grammar, the grammar was revised and enhanced to account for desired functionality and to correct problems. The language definition includes the individual token descriptions for the grammar primitives. A tokenizer state machine is also provided that defines the interpretation of input files into tokens.

## X10Gimli Grammar Definition

*<x10gimli>* ::= **x10gimli** *<ident:new>***;** *<x10gimli-env>* *<modes>* **end x10gimli.**
*<x10gimli-env>* ::= *<imports>* *<definitions>* *<functions>* *<triggers>* *<start-block>* *<controls>* *<finish-block>*

*<imports>* ::= *<null>* | **imports** *<import-list>*
*<import-list>* ::= *<null>* | *<import>* *<import-list>*
*<import>* ::= *<value:string>***;**

*<definitions>* ::= *<null>* | **definitions** *<definition-list>*
*<definition-list>* ::= *<null>* | *<definition>* *<definition-list>*
*<definition>* ::= *<ident:new>* **=** *<values>***;**

*<functions>* ::= *<null>* | **functions** *<function-list>*
*<function-list>* ::= *<null>* | *<function>* *<function-list>*
*<function>* ::= *<ident:new>* *<idents>* *<command>*

*<triggers>* ::= *<null>* | **triggers** *<trigger-list>*
*<trigger-list>* ::= *<null>* | *<trigger>* *<trigger-list>*
*<trigger>* ::= *<ident:new>* **(***<values>***) do** *<command>* | **(***<values>***) do** *<command>*

*<modes>* ::= *<mode-list>*
*<mode-list>* ::= *<null>* | *<mode>* *<mode-list>*
*<mode>* ::= **mode** *<ident:new>* *<x10gimli-env>*

```
<start-block>       ::= <null> | start <command>
<finish-block>      ::= <null> | finish <command>

<controls>          ::= <control-list>
<control-list>      ::= <null> | <control> <control-list>
<control>           ::= control <<switch-value-list>> <switches>

<switches>          ::= <switch-list>
<switch-list>       ::= <null> | <switch> <switch-list>
<switch>            ::= [<switch-value-list>] <command>
<switch-value>      ::= <variable>
<switch-value-list> ::= <switch-value> | <switch-value>, <switch-value-list>

<command-list>      ::= <null> | <command> <command-list>
<command>           ::= <func-command> | <if-command> | <while-command> | <ident-command> |
                        <trigger-command> | <motion-command> | <begin-command> | <return-
                        command>

<begin-command>     ::= begin <command-list> end
<func-command>      ::= <function-call>;
<if-command>        ::= if <values> then <command> <else-command>
<else-command>      ::= <null> | else <command>
<while-command>     ::= while <values> do <command>
<trigger-command>   ::= trigger <trigger>
<ident-command>     ::= <variable> = <values>; | <variable>;
<motion-command>    ::= motion <variable>; <start-block> <finish-block> | motion <variable>
                        <variable>; <start-block> <finish-block>
<return-command>    ::= return <variable>;
<define-command>    ::= define <definition>;
<new-command>       ::= new <definition>;

<function-call>     ::= <ident:func> <params>

<values>            ::= <variable> <value-list>
<value-list>        ::= <null> | <symbol:operator> <values> | <symbol:condition> <values> |
                        <symbol:logic> <values>

<params>            ::= (<param-list>) | ()
<param-list>        ::= <param>, <param-list> | <param>
<param>             ::= <variable>

<idents>            ::= (<ident-list>) | ()
<ident-list>        ::= <ident:new>, <ident-list> | <ident:new>

<variable>          ::= <ident:new><variable-item> | <value> | (<values>)<variable-item> |
                        <ident:predefined><variable-item> | <function-call> |
                        {<variables>}<variable-item> | <ident:new><params>
<variables>         ::= <null> | <param-list>
<variable-item>     ::= <null> | [<variable>] | .<variable>
```

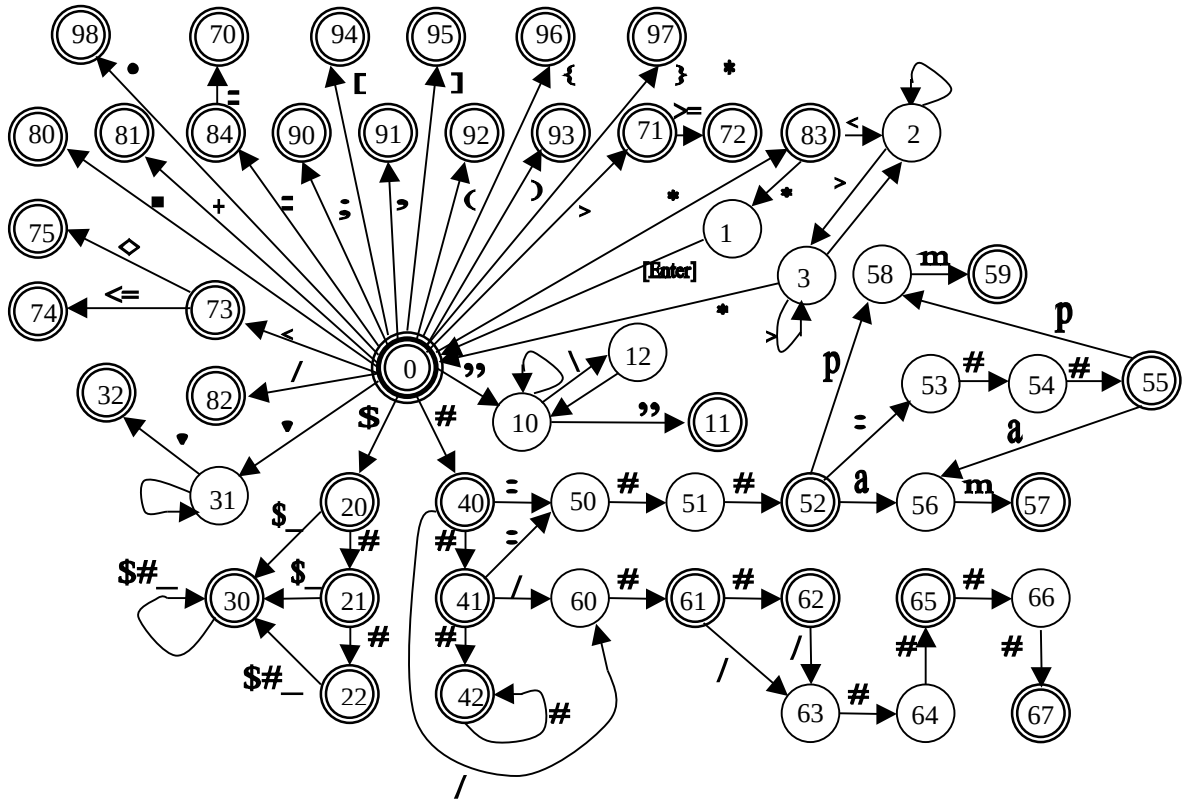
## Tokenizer Data Types and Formats

| | |
|---|---|
| *<ident:func>* | ::= **x10switch** \| **toswitch** \| **tocontrolswitch** \| **tocontrol** \| **toggle** \| **log** \| **sound** \| **x10address** \| **x10command** \| **activate** \| **deactivate** \| **packet** |
| *<ident:predefined>* | ::= **time \| code** \| **command** \| **address** \| **day** \| **month** \| **date** \| **inputpacket** |
| *<ident:new>* | ::= $+[**$#_...$#_**] \| **'...'** |
| *<symbol:operator>* | ::= **+** \| **-** \| ***** \| **/** |
| *<symbol:condition>* | ::= **>** \| **>=** \| **<** \| **<=** \| **<>** \| **==** |
| *<symbol:logic>* | ::= **and** \| **or** |
| *<value:number>* | ::= [#...#] |
| *<value:x10>* | ::= **$##** \| **$#** \| **$** |
| *<value:string>* | ::= **"..."** |
| *<value:time>* | ::= **##:##am** \| **##:##pm** \| **##:##** |
| *<value:day>* | ::= **sun** \| **sunday** \| **mon** \| **monday** \| **tue** \| **tuesday** \| **wed** \| **wednesday** \| **thu** \| **thursday** \| **fri** \| **friday** \| **sat** \| **Saturday** |
| *<value:month>* | ::= **jan** \| **january** \| **feb** \| **february** \| **mar** \| **march** \| **apr** \| **april** \| **may** \| **jun** \| **june** \| **jul** \| **july** \| **aug** \| **august** \| **sep** \| **september** \| **oct** \| **october** \| **nov** \| **november** \| **dec** \| **december** |
| *<value:comm>* | ::= **allunitsoff** \| **alllightsoff** \| **alllightson** \| **on** \| **off** \| **dim** \| **bright** |
| *<value:date>* | ::= **##/##** \| **##/##/####** |
| *<value:boolean>* | ::= **true** \| **false** |
| *<value:list>* | ::= **{...,...}** |
| *<value:item>* | ::= *<variable>***[***<variable>***]** \| *<variable>***.***<variable>* |
| *<comment>* | ::= ***<...>*** \| ****...[Enter]** |

## Tokenizer State Descriptions

The X10Gimli input files are converted into individual tokens before the parser can handle it. The token scanner is implemented using the state machine that is described below. Input characters make the transitions between individual states. Tokens can follow on after another with no white space, as defined by the scanner.

All undefined transitions preserve the current input character and proceed to a final token state, either valid or invalid. The valid and invalid states create the actual tokens based on the input. Final states are designated by a double circle and have an unshown transition to a valid state, while intermediate state have a transition to an invalid state. The start state also has a transition to an invalid state for unhandled input. The valid state is only entered from final states and uses the final state as a reference to create the new token. The invalid state creates an error token.

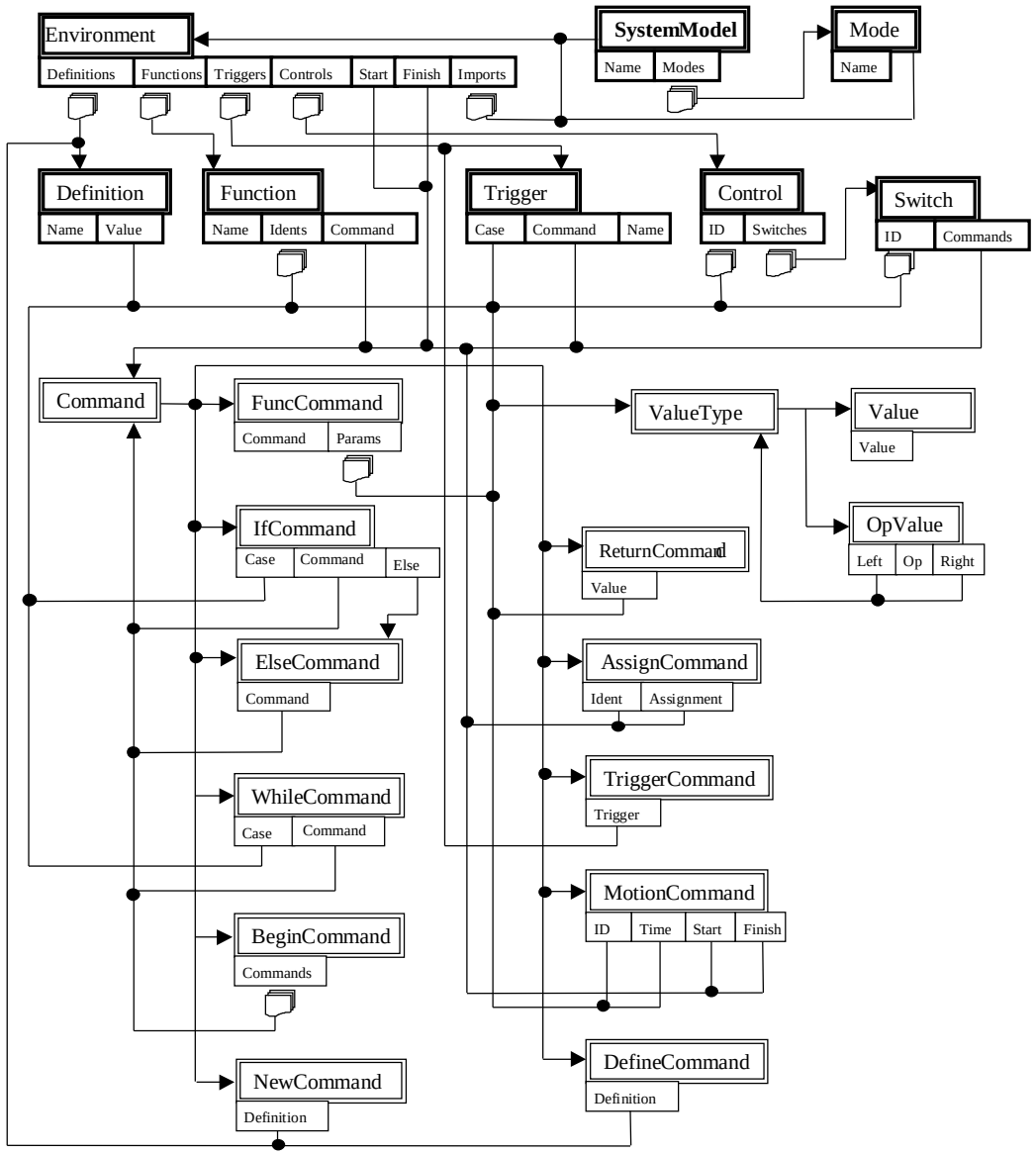



The numbered states in the diagram represent different token paths as described below. Bold state numbers represent valid input states.

| | |
|---|---|
| 0 | start state for all tokens |
| 1 | comment - **... |
| 2 | block comment - *<... |
| 3 | block comment - *<...> |
| | |
| 10 | string - “... |
| **11** | string - “...” |
| 12 | string - escape code |
| | |
| **20** | x10 - $ |
| **21** | x10 - $# |
| **22** | x10 - $## |
| | |
| **30** | $[$#_] \| $#[$#_] \| $##[#$_] |
| 31 | identifier - ‘... |
| **32** | identifier - ‘...’ |
| | |
| **40** | number - # |
| **41** | number - ## |
| **42** | number - ### |
| | |
| 50 | time - ##**:** \| #**:** |
| 51 | time - ##**:**# \| #**:**# |
| **52** | time - ##**:**## \| #**:**## |
| 53 | time - ##**:**##**:** \| #**:**##**:** |
| 54 | time - ##**:**##**:**# \| #**:**##**:**# |
| **55** | time - ##**:**##**:**## \| #**:**##**:**## |
| 56 | time - ##**:**##**:**##**a** \| #**:**##**:**##**a** \| ##**:**##**a** \| #**:**##**a** |
| **57** | time - ##**:**##**:**##**am** \| #**:**##**:**##**am** \| ##**:**##**am** \| #**:**##**am** |
| 58 | time - ##**:**##**:**##**p** \| #**:**##**:**##**p** \| ##**:**##**p** \| #**:**##**p** |
| **59** | time - ##**:**##**:**##**pm** \| #**:**##**:**##**pm** \| ##**:**##**pm** \| #**:**##**pm** |
| | |
| 60 | date - ##/ \| #/ |
| **61** | date - ##/# \| #/# |
| **62** | date - ##/## \| #/## |
| 63 | date - ##/##/ \| #/##/ \| ##/#/ \| #/#/ |
| 64 | date - ##/##/# \| #/##/# \| ##/#/# \| #/#/# |
| **65** | date - ##/##/## \| #/##/## \| ##/#/## \| #/#/## |
| 66 | date - ##/##/### \| #/##/### \| ##/#/### \| #/#/### |
| **67** | date - ##/##/#### \| #/##/#### \| ##/#/#### \| #/#/#### |
| | |
| **70** | == |
| **71** | > |
| **72** | >= |
| **73** | < |
| **74** | <= |
| **75** | <> |
| | |
| **80** | - |
| **81** | + |
| **82** | / |
| **83** | * |
| **84** | = |
| | |
| **90** | ; |
| **91** | , |
| **92** | ( |
| **93** | ) |
| **94** | [ |
| **95** | ] |
| **96** | { |
| **97** | } |
| **98** | . |

# System Architecture

The X10Gimli system is structured according to the following diagram (highlighting the main system classes), which was designed based on the X10Gimli grammar:




Each of the highlighted components in the diagram represents an integral part of the system. The actual component pieces will described further.

# System Model

The X10Gimli system model is the control center for input processing and code execution. The system model is an extension and enhancement of the expanded environments. The model contains the list of modes, and does all real-time execution, packet handling, and trigger checking.

Worth noting, and related the system as a whole, is the use of Java reflection to enhance the language. X10Gimli is intended to be a point of control for many types of devices and software. The language itself cannot access external resources. However, external resources can be accessed by means of user defined Java classes and native code. Static variables, functions, and interface gateways can be used within the language by following some conventions that are defined for the related features. For example, if one wanted access to real-time temperature information, a pre-defined identifier called `TEMPERATURE` could be created. The temperature information could come from the internet, a house sensor, or other location external to X10Gimli and imported by the user's identifier. Anytime that an identifier is evaluated during execution, it would return the current temperature.

The model stores a list of input and output gateways. The input gateways are the source of all input into the model. For example, the MR26A receives RF X10 input, but cannot transmit. Output gateways transmit data from the model to different devices or processes, such as the CM11A, which can transmit X10 signals over the power lines. Gateways, can be both input and output Gateways, like the CM11A. Interface gateways are activated in source code and can be done within the model's start command. For instance:

```
x10gimli interfaces;
start
   begin
      interface(UDP, 3423);
      interface(CM11A, "COM1", "X10A");
      interface(MR26A, "COM2", "X10B");
   end
```

As is shown, when interfaces are activated, they can take an initialization parameter. The parameter can be a list if necessary. The other interface parameter renames the gateway. Different gateway names can be useful for distinguishing between input sources as well as specifying specific destinations for output packets.

All input received by the model is placed in a queue. The queue input is handled in a FIFO fashion by an input handling thread. That way, no input is lost and things can be processed in order. Input handling is done by finding all controls that could possibly respond to the input by searching the active mode and visible environments. The valid controls are then searched in order for a suitable switch for the input. The first valid switch found is then executed.

A separate thread does trigger checking and all triggers are checked at regular intervals. When a trigger or group of triggers fires, the execution of those triggers is handled in a new thread. This is done so regular trigger checking can continue.

## Expanded Environment

Full X10Gimli environments possess information beyond what small execution environments provide. Expanded environments contain user function definitions, permanent trigger lists, X10Gimli controls, and temporary triggers. Start and finish commands can be executed when transitioning into and out of an execution environment. Transitioning is done by the X10Gimli system when changing modes.

The execution environment at this level can also extend through imported environments. When evaluating identifiers, and checking controls, triggers, and functions during run-time, imported environments are checked before parent environments. This allows for imported definitions, functions, controls, and so on.

Here is an example of extending environments within a mode:

```
mode Ex
   imports
      "defA.GIM";
   definitions
      varB = varA;
   control <true>
      [true]  log ("varB = "+varB+" which equals varA.");
```

In this example, assume that "defA.GIM" has `varA` defined to equal 6. When the control switch gets executed, the output will be:

```
"varB = 6 which equals varA."
```

This is because the imported environment is used when evaluating expressions, while is also acts as an extended portion of the local environment for everything else.

## Basic Environment

Small individual environments determine the scope of evaluation for all X10Gimli interpretation. The execution scope is determined by the tree structure created by the environment ancestors. Each environment has a list of local variables that are used in evaluating expressions. Every environment also has a parent environment that is deferred to when a definition is not found. This is done until the root node is reached.

Also, each environment can return a value to parent environments. In this way, functions can return values. An example of an environment tree follows:

```
x10gimli env;
definitions
   defa = 1;
mode modea
   definitions
      defb = defa;
   start
      begin
         defa = 2;
```

```
            defc = defb;
            begin
               log ("DefC = "+defc+".");
            end
         end
end x10gimli.
```

In the preceding full example, the resulting output will be:

```
"DefC = 2."
```

The environment tree at the point of log execution starts at the system model root, and continues to the mode, to the begin block of the start statement, and to the inner begin block. The `defa` variable is in the visible scope and is reassigned. When `defb` is evaluated, `defa` is still in scope, so the value of `DefC` is 2.

## Import Manager

All files loaded under the X10Gimli system are managed by the import manager. This system component allows a few important things to happen when handling X10Gimli source files.

Since the import manager keeps a list a desired input paths, import file names can be automatically resolved using the given paths. That way, source files can also import other files without having to explicitly state the import file paths. The default path is the current directory, or a full path.

Managing all loaded files in one location also allows an environment web to be created where more than one path exists to any variable, function, control, or switch. In that way, common variables can be shared between X10Gimli environments. For example:

**COMMON.GIM**

```
x10gimli common;
definitions
   varcommon = a;
end x10gimli.
```

**IMPA.GIM**

```
x10gimli impa;
imports
   "COMMON.GIM";
control <INPUT>
   [(varcommon == a)] log("common == a);
end x10gimli.
```

**IMPB.GIM**

```
x10gimli impa;
imports
   "COMMON.GIM";
control <INPUT>
   [(varcommon == b)] log("common == b);
end x10gimli.
```

In the example above, the two files "IMPA.GIM" and "IMPB.GIM" access the same common variable. The variable can be accessed commonly to the two environments because of the way imported files are managed by the import manager.

## Definition

In X10Gimli a definition is synonymous to a variable identifier. A definition uses a text tag to associate X10Gimli values to identifier names. Definitions can store unevaluated expressions that are only evaluated during run-time, and literal values.

Unevaluated expressions can be used to allow multiple levels of variable indirection. For example, an X10Gimli source file could contain the following:

```
define def1 = 5;
define def2 = def1;
def1 = 6;
if def2 == 6 then log ("def2 is using unevaluated value
information");
```

Because the definition `def2` stores the unevaluated expression “def1”, when it comes time to get the usable value of `def2`, its value is 6. That's because defined expressions are only fully evaluated during runtime. Another way to do unevaluated definitions is using the `definitions` block in an X10Gimli program, such as:

```
definitions
   def1 = 5;
   def2 = def1;
```

It is also possible to do fully evaluated assignments, as in the following example:

```
define def1 = 5;
def2 = def1;
def1 = 6;
if def2 == 5 then log ("def2 is using evaluated value
information");
```

The functioning of variables in X10Gimli is flexible in that way, and the definitions store the necessary information.

## Function

X10Gimli functions can be defined by users in source files. Such functions can be called during run-time just like any other function. Users define the parameters that are used in the function and those parameters are mapped as local identifier definitions in the command block of the function. When parameter identifiers are evaluated, the local definitions are applied and that's how the passed in parameters get applied to the local commands. Since a function is just an extension of a normal execution environment, it can return values to parent environments, and it's scope is a continuation of previous environments. For example:

```
functions
   addx(varx) return varx + inx;
   func(inx) return addx(10);
```

Note that in the example, the variable `inx` is referenced from within the `addx` function. This can be done because `addx` is called from `func`, and, therefore, has that function's parameters in its scope.

## Trigger

Triggers in X10Gimli are checked periodically and when certain inputs occur. Triggers allow resultant events to occur due to time, date, input received, and the conditional state of the environment.

The X10Gimli model periodically checks to see if any triggers can fire. When a trigger fires, its command statement is executed, and the trigger is reset. Depending on the conditional types in a trigger, individual triggers can only fire after a certain amount of time has elapsed. The reset delays for triggers that contain specific types of value comparisons is given below:

- TIME : 1 minute
- DAY : 1 day
- MONTH : 1 day
- DATE : 1 day

Other types are immediately available to be triggered again. Note, however, that local triggers defined with the `trigger` command can only be fired once, and are then lost. Here is an example of an X10Gimli trigger:

```
triggers
   late (time == 11:00pm) do turnLightsOff();
```

This example defines a trigger that will fire at 11:00pm.

## Mode

X10Gimli modes contain information regarding separate execution environments. The X10Gimli model sends inputs to the active mode when it needs to be handled. All modes have names and contain the environment information necessary to exist separately from other modes. The root environment of all modes should is the system model in X10Gimli. Here is an example of mode definitions in the language:

```
mode ModeA
   control <X10>
      [A10]  log ("ModeA control");

mode ModeB
   control <X10>
      [A10] log("ModeB control");
```

In the example, the X10 input `A10` will print a different message depending on the current active mode. The input doesn't change, but the reaction is different because of different modes.

## Control

In X10Gimli a control defined in source code can be compared to a physical remote control. As with physical remote controls, an X10Gimli control processes input and possibly sends output. Switches within each control define the input that can be processed, much like how a physical remote control has separate buttons that can cause different things to happen. Following that idea, X10Gimli controls are defined.

So, in X10Gimli, individual controls are specified to respond to various inputs. Each control is related to a specific type of input and/or condition in the X10Gimli environment. Input types are varied and are determined by the originating devices and processes, such as X10 interfaces or RF remotes.

Controls contain all of the switches that process input. A switch is made up of a set of identification values and input responses. Controls allow individual switches to be grouped into a single categorized location. Using X10Gimli source code, controls can be defined as shown:

```
control <MEDIACONTROL, (RF_MODE == RF_WINAMP)>
```

Note that the control identifier values will be evaluated during runtime, and the evaluated identifiers are used when checking input. For example, `(RF_MODE == RF_WINAMP)` will evaluate to true or false. When the input type matches the control and all conditions are true, then the switches for hat control will be checked.

## Switch

In X10Gimli a switch can be compared to one of the buttons on a physical remote control. When a remote control button is pressed, resulting action occurs, such as changing the remote's mode to VCR, or sending an IR signal to change channels. In a similar fashion X10Gimli switches have identification and commands that can be executed when the switch is triggered. Also, X10Gimli switches can have an enabled or disabled state. In source code, switches can be defined within controls as shown:

```
control <MEDIACONTROL, (RF_MODE == RF_WINAMP)>
   ["PAUSE", (WINAMP_PAUSE == true)] begin doPlay(); end
   ["PAUSE", (WINAMP_PAUSE == false)] doPause();
```

Note that the switch identifier expressions will be evaluated during runtime, and the evaluated identifiers are used when checking input. For example, `(WINAMP_PAUSE == true)` will evaluate to true or false. The commands that follow make up the switch's response.

# Packet

X10Gimli packets are the main structure for passing information to and from the X10Gimli system. Packet are structured as follows:

- X10Gimli header (**9 bytes**)
- flags (**1 byte**)
- reserved [not used](**4 bytes**)
- ID number (**4 bytes**)
- packet size (**2 bytes**)
- type string (**20 bytes**)
- source string (**20 bytes**)
- destination string (**20 bytes**)
- number of values (**1 byte**)
- list of tags and values (**variable size**)

Packets have two internal data representations – packed and unpacked – which are tracked and handled by the packets. The packed data format consists of a byte array. Byte arrays can be easily transmitted over network sockets, received, and unpacked in other locations or in other processes.

In the X10Gimli system, packets are interpreted as switch input to specified controls. The packet's type string represents the control it is intended for. The tagged value list represents switch information. When a packet is received in the system model, a control identification list is created using the type string. A switch identification packet is created using all of the tagged values. For example, a packet with the type string "X10", and tagged values `HOUSECODE=A` and `DEVICECODE=15`, could be handled by any of the following switches:

```
control <X10>
   [A, 15]  log(""+housecode+devicecode);
   [A] log(""+housecode);
   [15] log(""+devicecode);
```

Any of the above switches valid since a switch only must find a match for each of its identification values. However, in this case, only the first switch in the list would be triggered. The example also demonstrates that tag names in incoming packets become variable identifiers that can be used in expressions.

X10Gimli also provides a way for packets to be created internally. Internal packet values store type and tag information, but not network transit data. An internal packet example is given:

```
begin
   vpack = Packet ("X10", {HOUSECODE, DEVICECODE}, {A, 15});
   vpack.HOUSECODE = B;
   vpack.DIMS = 55;
   vpack.TYPE = "X10NEW";
end
```

As is shown, tagged values can be accessed in internal packets. New tags can also be added by assigning values to non-existent tag names. The packet type can be changed as well.

This is the first version of the X10Gimli packet class. As is shown in the structure description, there are a few fields that can store data, but are not currently used by the system. Future revisions to the packet will take advantage of the available bytes.

# Standard Data Types

The X10Gimli data types make up all of the values within the language. Individual data types each have a unique set of legal operations that can be performed between them and other data types. The available data types are given below:

- **Ident** - plain variable identifier
- **Number** - integer number
- **X10** - X10 housecode/devicecode combination
- **String** - string
- **Time** - time in hours, minutes, and seconds
- **Day** - day of the week
- **Month** - month of the year
- **Command** - X10 command
- **Date** - date
- **Boolean** - boolean
- **List** - list of values
- **Packet** - contains tagged values

The compatible operations between different data types is shown in the chart below. Each row shows which operations can be performed. The rows represent the left operand, and the columns represent the right operand. There is no current support for unary operators.

**Operator Evaluation Chart**

| | Ident | Num | X10 | String | Time | Day | Month | Comm | Date | Bool | List |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Ident | + == > >= <> < <= | + | + | | | + | + | + | | | |
| Num | | +-*/ == > >= <> < <= | +-*/ == > >= <> < <= | | | +-*/ == > >= <> < <= | +-*/ == > >= <> < <= | +-*/ == > >= <> < <= | | | |
| X10 | | +- | +- == > >= <> < <= | | | | | | | | |
| String | + | + | + | + == > >= <> < <= | + | + | + | + | + | | |
| Time | | +- | | | +- == > >= <> < <= | | | | | | |
| Day | | +- == > >= <> < <= | | | | +- == > >= <> < <= | | | == > >= <> < <= | | |
| Month | | +- == > >= <> < <= | | | | | +- == > >= <> < <= | | == > >= <> < <= | | |
| Comm | | +- == > >= <> < <= | | | | | | +- == > >= <> < <= | | | |
| Date | | +- | | | | +- == > >= <> < <= | +- == > >= <> < <= | | +- == > >= <> < <= | | |
| Bool | | | | | | | | | | AND OR == | |
| List | + | +*/ | + | + | + | + | + | + | + | + | + |

No operators can be directly applied to packet values.

Expressions in X10Gimli have no operator precedence. Expressions are evaluated from left to right. The left operand determines the resultant data type for any evaluation. Normally, the resultant value will be of the type of the left operand. Any evaluation will proceed as described. For example, to create a string from a number, simply add the number to a string:

```
""+423
```

With that, the string "423" is created due to the order of evaluation. The same does not apply in reverse:

```
423+""
```

The second example would generate an evaluation error because a string cannot be added to a number, per the operator evaluation chart. Some data types have unique operator relationships that are explained in the individual value descriptions.

# Boolean

*Boolean* data type. Boolean values are returned from any comparison operation. They are the basis for all conditional expressions. All lookups for controls and switches use a `true` boolean value when comparing the evaluated expressions.

**Internal Values:**

Stores a `true` or `false` value.

**Operators Supported:**

**AND** (*Boolean*)

Performs a logical `and` operation.

**OR** (*Boolean*)

Performs a logical `or` operation.

# Command

X10 command data type. Incoming X10 commands and outgoing commands are represented by this data type.

**Internal Values:**

Stores one of the 16 possible X10 function values (listed in numerical order, starting at 0):

- allunitsoff
- alllightson
- on
- off
- dim
- bright
- alllightsoff
- extendedcode

- hailrequest
- hailacknowledge
- presetdim1
- presetdim2
- extendeddatatransfer
- statuson
- statusoff
- statusrequest

**Operators Supported:**

+, - (*Command, Number*)

Loops circularly through the command list by adding or subtracting a numerical value.

>, <, >=, <=, ==, <> (*Command, Number*)

Performs a comparison based on the numerical representation of the command value.

## Date

*Date* data type. Dates can be used to allow special and very specific trigger and switch responses to date conditions. For example, special things could be made happen on somebody's birthday by using the date value.

**Internal Values:**

Stores a calendar with the day, month, and year.

**Operators Supported:**

+, - (*Number, Day*)

Adds or subtracts a certain number to the date.

+, - (*Month*)

Adds or subtracts a certain number of months.

+, - (*Date*)

Adds or subtracts all of the date fields.

>, <, >=, <=, ==, <> (*Number, Day*)

Does a comparison on the date, not comparing month or year.

>, <, >=, <=, ==, <> (*Month*)

Does a comparison on the month only.

>, <, >=, <=, ==, <> (*Date*)

Does a comparison on the day, month, and year.

## Day

*Day* data type. Inividual days of the week can be handled with this data type. It is possible that unique triggered events and controls are desired for different days of the week. For example, if sprinklers are only wanted on Monday, Wednesday, and Friday, such functionality can be acheived with this data type.

**Internal Values:**

Stores a number representing the day of the week, where Sunday is 1.

**Operators Supported:**

+, - (*Number*, *Day*)

Adds or subtracts a certain number to the day in a circular fashion.

>, <, >=, <=, ==, <> (*Number*, *Day*)

Does a comparison based on the numerical day of the week value, where Sunday is 1.

>, <, >=, <=, ==, <> (*Date*)

Does a comparison on the day of the week numerical value.

## Ident

*Ident* data type. All variables in the language are accessed by way of the identifier data type. The identifier string is used to dereference definitions, and definitions will always be evaluated before evaluating expressions. However, identifier values that do not map to any existing definition can have opertations applied directly to them to create new unique identifiers.

**Internal Values:**

Stores a string identifier.

**Operators Supported:**

+ (*Ident*, *Number*, *X10*, *Day*, *Month*, *Command*)

Appends the string representation of the value to the identifier.

>, <, >=, <=, ==, <> (*Ident*)

Does a string comparison on the identifiers.

## List

*List* data type. The list data type functions similar to an array, but has no type or size restrictions. Arithmetic operators have special functionality with lists. Values can be accessed within lists by referencing them starting with index value 1. A list can be resized by reassigning the value at index 0.

**Internal Values:**

Stores a list of values.

**Operators Supported:**

+ (*Ident*, *Number*, *X10*, *Day*, *Month*, *Command*, *String*, *Time*, *Date*, *Boolean*)

Adds the item to the end of the list.

+ (*List*)

Appends two lists together to make a new list.

/ (*Number*)

Deletes an element from the list at the specified index.

***** (*Number*)

Inserts an entry in the list at the specified index. The value must be set afterwards.

## Month

*Month* data type. Explicit month of the year values allow for advanced handling. For example, if someone wanted holiday music automatically loaded into an MP3 player during December, a month based trigger could be used.

**Internal Values:**

Stores a numerical month value, where January is 1.

**Operators Supported:**

+, - (*Number*, *Month*)

Adds or subtracts a certain number to the month in a circular fashion.

>, <, >=, <=, ==, <> (*Number*, *Month*)

Does a comparison based on the numerical month value, where January is 1.

>, <, >=, <=, ==, <> (*Date*)

Does a comparison on the month numerical value.

## Number

*Number* data type. Integer numbers are extremely important in any language. No floating point support is currently available in the system, but integer numbers allow for sufficient flexibility to accomplish many useful tasks.

**Internal Values:**

Stores an integer number.

**Operators Supported:**

+, -, *****, / (*Number*, *Day*, *Month*, *Command*, *X10*)

Does an addition, subtraction, multiplication, or division operations between the numerical representations of these values.

>, <, >=, <=, ==, <> (*Number*, *Day*, *Month*, *Command*, *X10*)

Does comparison between the numerical representations of these values.

## String

*String* data type. Strings essentially can represent any kind of data. String value representations can be parsed and used in many ways. X10Gimli allows string arithmetic to be performed, where any data type is converted into a valid X10Gimli string representation.

**Internal Values:**

Stores a list of values.

**Operators Supported:**

+ (*Ident*, *Number*, *X10*, *Day*, *Month*, *Command*, *String*, *Time*, *Date*, *Boolean*, *List*)

Appends the string representation of the value to the existing string.

>, <, >=, <=, ==, <> (*String*)

Does a string comparison between the string values.

## Time

*Time* data type. Time values are critical for the functionality of triggers. Motion triggers are based on time delay. Time based events, such as turning on lights and controlling sprinklers, require time value comparisons. The time data type allows for a certain degree of precsion for comparisons, using minutes or seconds. Comparison precision is determined by the amount of precision used to initially describe the value.

**Internal Values:**

Stores the time with the hour, minute, and second.

**Operators Supported:**

+, - (*Number*)

Adds or subtracts a certain number of seconds to the time.

+, - (*Time*)

Adds or subtracts a the hours, minutes, and seconds in a circular fashion.

>, <, >=, <=, ==, <> (*Time*)

Does a comparison on the time.

## X10

*X10* address data type. X10 values can be only a house code, or a house code and device code combinaiton. Since X10Gimli was initially intended to allow advanced control of X10 devices, the X10 address data type is naturally available.

**Internal Values:**

Stores an X10 housecode and devicecode.

**Operators Supported:**

+, - (*Number*, *X10*)

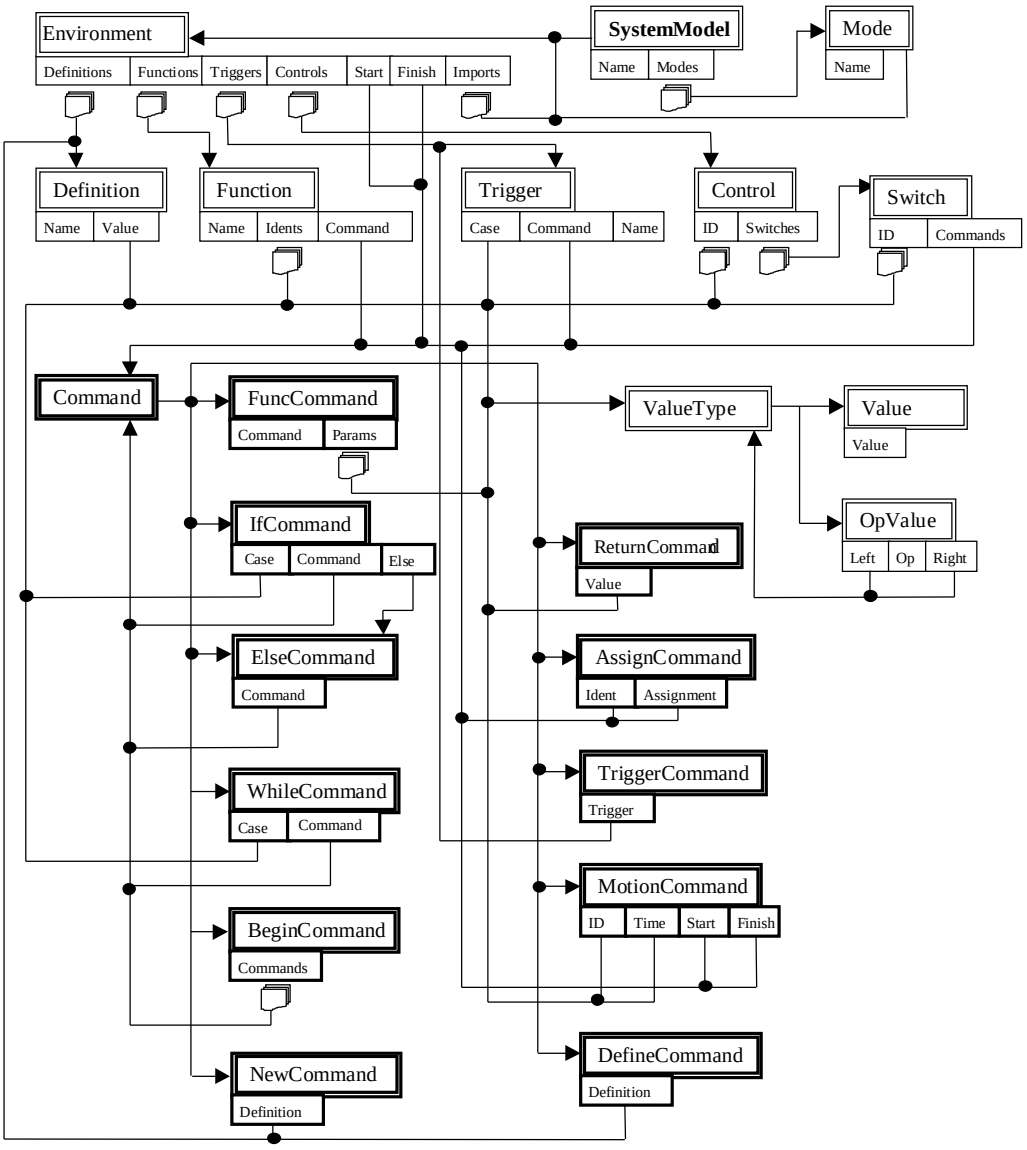
Adds or subtracts from the device code number in a circular fashion.

>, <, >=, <=, ==, <> (*X10*)

Does a comparison on the housecode and devicecode. A devicecode can be greater than or less than another only when the housecodes are equal. Otherwise, not equal is produced.

# Standard Language Statements

The language provides several commands structures that define how execution is performed. These commands are outlined in the language grammar. Each command statement is described in detail below.



# Define Statement

<*define-command*> ::= **define** <*definition*>**;**

This command is used to handle variable creation at the expanded environment level, and to do unevaluated expression assignments. The system model and individual modes constitute expanded environments. If no matching definition is found in visible scope, then a new definition is created at in the expanded environment, or in other words, at a broad scope level. If a definition already exists that matches the desired definition, then its value is reassigned to the one specified by this command. The value assigned is an unevaluated expression. For example:

```
begin
   varu = 10;
   varx = 5;
   begin
      vary = 6;
      begin
         varz = 4;
      end
      vary = varz;
      varx = vary;
   end
   vary = 23;
   begin
      define vary = varu-10;
      define varz = varu;
   end
   varu = 15;
   log (""+varx+", "+vary);
end
```

The output in this case would be:

```
"15, 5"
```

In this example the important effects of this command are illustrated:

- Following scope rules, at the last value assigned to `varx` is the value of `vary`. Because `varz` was not in scope when `vary` was previously assigned a value, `varx` is assigned the identifier `varz`. Later, `varz` is defined at a deeper point of scope, but the `define` command causes it to be defined at a more global point with an unevaluated identifier `varu`. Therefore, when the value of `varx` is printed, it evaluates through `varz` through `varu` to the current value of `varu`, which is 15.
- The case of printing the value of `vary` follows the similar pattern. In this case, `vary` is defined with an unevaluated expression at a deeper point of scope. When it is finally evaluated, its value is shown to be the current value of `varu` minus 10, which is 5.

In general, new definitions should not need to be added outside of the `definitions` block of an `X10Gimli` source files, but the capability does exist. It is handy, however, to be able to assign values of unevaluated expressions and identifiers to other variables.

**Usage:**

**define** *identifier* **=** *expression*;

## New Statement

<*new-command*> ::= **new** <*definition*>**;**

This statement creates new variables in the local environment. The `new` statement differs from the assignment and `define` statements because under all circumstances, the `new` command adds a new definition to the local environment. For example:

```
x10gimli test;
functions
   loopfunc1()
      begin
         loop = 3;
         while (loop > 0) loop = loop - 1;
      end
   loopfunc2()
      begin
         new loop = 3;
         while (loop > 0) loop = loop - 1;
      end
start
   begin
      loop = 5;
      loopfunc2();
      varx = loop;
      loopfunc1();
      vary = loop;
      log (""+varx+", "+vary);
   end
end x10gimli.
```

In this case, the output will be:

```
"5, 0"
```

This example demonstrates some of the effects of scope in X10Gimli and how to properly deal with it.

- Both of the function calls execute the same code, but the effect is not quite the same. In `loopfunc1` the loop counter is set, but the variable being set is not actually local. This can cause unexpected side effects if the code is not designed with this in mind. In `loopfunc2` the `new` statement is used to declare a local variable for the function. This should be done for most functions using local variables.

So, this command allows unevaluated variable definitions to be created in the local environment space.

**Usage:**

**new** *identifier* **=** *expression*;

## Identifier Assignment Statement

<*ident-command*> ::= <*variable*> = <*values*>**;** | <*variable*>**;**

This statement handles assigning new values to variable definitions. New values can be assigned to identifiers, packet tags, and list elements. If there is no existing definition for a given identifier, then a new identifier will be created within the local execution environment and assigned the value. It is also important to note that the value being assigned is the full evaluation of an expression. Here is an example of the assignment command being used in many ways:

```
define varx = 5;
define vary = {2, 3, 4};
define varz = Packet(TEST, {IDA, IDB}, {a, b});

varx = vary[2];
vary[2] = 7;
vary[0] = 4;
vary[4] = var[3] + 23;
varz.IDA = varz.IDA+3;
varz.IDC = c;
begin
   varw = 5;
end
varu = varw;
varw = 7;
varu = varu + 1;
Log ("Variables = "+varx+", "+vary+", "+varz+", "+varu);
```

In this example, the output would be:

```
 "Variables = 3, {2, 7, 4, 27}, PACKET(TEST, {IDA, IDB, IDC}, {A3,
B, C}), 8"
```

This illustrates several important regarding assigning values.

- Notice that the value of `varx` is still 3 after `vary[2]` has been changed. This is because the assigned value is fully evaluated. The `define` command assigns unevaluated expressions to identifiers.
- Note that assigning a new number value to the list element 0 changes the size of the list.
- Notice that assigning a value to a packet tag that does not exit adds the tag and value to the packet.
- Notice that the `varw` identifier was assigned a value in a different scope. When `varu` is assigned the `varw` value, there is no `varw` identifier in scope, so `varu` is assigned the plain identifier `varw` instead of an evaluated expression. Another `varw` is defined in visible scope afterwards, and when `varu` adds 1 to itself, it is actually adding one to `varw`, and so the value becomes 8, not 6.

In general, when using assignment statements, it is important to remember that the expression is fully evaluated before being assigned to the identifier.

**Usage:**

```
variable = expression;
```

## Begin/End Block

<*begin-command*> ::= **begin** <*command-list*> **end**

This command is used to perform the execution of a block of command statements. All pieces of the X10Gimli system deal with single commands. By encapsulating multiple commands within the `begin` and `end` keywords, command blocks are created. This command also spawns a new execution environment and extends the visible scope. Upon completion of the command block, any return value in the spawned environment is itself returned. Here is an example of using this command:

```
begin
   varx = 5;
   if (varx == 5) then begin
      varx = varx + 2;
      vary = 3;
   end
   if (vary == 3) then varx = 2;
   varz = 0;
   while (varx > 0) do begin
      varx = varx - 1;
      vary = vary + varx;
      varz = vary;
   end
   log (""+varz);
end
```

The output of this example would be:

```
"VARY7654321"
```

This example presents command blocks that handle multiple command execution, as well as the effects of scope, and operator evaluation within the different blocks.

- The variable `vary` is created within a command block, and is only available within that scope. When the command block steps out, `vary` is lost. The next statement attempts to do a comparison between an identifier and a number. Such a comparison is illegal because the identifier `vary` does not evaluate to anything but itself. An error will be posted.
- During the while loop, `vary` will be assigned an unexpected value. Since it is not defined at the point it is first assigned a value, its assigned value will be the result of a number added to an identifier, which in the first case will be `VARY7`. The loop will repeat and continue to append numbers to varx, until the end.

Effectively using these block command statements requires understanding a little about how the scope functions, but it is otherwise straightforward.

**Usage:**

**begin** *commandlist* **end**

## If/Then Statement

<*if-command*> ::= **if** <*values*> **then** <*command*> <*else-command*>

X10Gimli `if` statements function the same as they would in just about any other language. `If` statements require an expression that evaluates to a boolean value. Expressions that evaluate to true cause the `if` response command to execute. False values cause an optional `else` command to execute. Here is an example:

```
begin
   varx = 5;
   if varx == 5 then vary = 6;
   if vary == 7 then begin end
   else if vary == 6 then
      log (""+vary);
end
```

The output from this example will be:

```
"6"
```

This example demonstrates one important thing to note about the execution responses to `if` conditions. In this example, `vary` is created in the response command of the `if` statement. This is possible because `if` response commands are executed in the same environment as the `if` evaluation.

**Usage:**

**if** *expression* **then** *command*;

## Else Statement

<*else-command*> ::= <*null*> | **else** <*command*>

This command simply executes another command when called  When an `if` statement is evaluated to false, the `else` command is executed instead. The only thing that happens is that another command is executed if one exists, as in this example:

```
begin
   varx = 10;
   if (varx == 9) then begin
   end else if varx == 11 then begin
   end else
      log (""+varx);
end
```

This example handles the execution of two `else` commands. The first command executes another `if` statement, which then causes another `else` to execute. The second command displays the output, which would be:

```
"10"
```

The `else` command can only be placed after and `if` statement.

**Usage:**

*`ifstatement`* **`else`** *`command`*

## While Statement

<*while-command*> ::= **while** <*values*> **do** <*command*>

This is the only loop command available in the X10Gimli system. Using a `while` loop, any other kind of loop can be simulated, so `for` loops, and `do/while` loops have not been added to the language. The usage of `while` loops is straightforward. Each loop requires a condition that determines whether to continue looping or not. Here is an example:

```
begin
   loop = 0;
   varx = "";
   while loop < 4 do
      begin
         varx = varx + vary;
         vary = loop;
         loop = loop + 1;
      end
   currenttime = time;
   while currenttime == time do
      varz = loop;
```

```
      if (varz > 0)
         log (""+varx);
   end
```

The output in this case will be:

```
"VARY012"
```

The provided example shows to special cases of using `while` loops.

- The execution of `begin` statements is handled in a special way. Normally `begin` commands create a new environment to execute in, which would mean the environment would be recreated for every step of the loop. However, the `while` creates an environment for the command block to execute in and preserves it until the loop terminates. So, in the first loop, `vary` isn't created until after the first assignment statement. The first addition is "" + VARY. After that, `vary` has a value and that value is appended. Under normal circumstance, there should be no need to create local variables that must exist during the duration of the loop, but the ability is available.
- Also, note that single commands executed in a loop are executed in the local environment. That allows new variables to be declared that remain in scope after the loop terminates. It is not likely that this adds any benefit, but it is possible.

The most important thing is to always remember is that the loop condition must become false at some point to terminate the loop. That usually requires incrementing a loop counter from within the loop.

**Usage:**

**`while`** *`expression`* **`do`** *`command`*`;`

## Return Statement

<*return-command*> ::= **return** <*variable*>**;**

This sets the return value for the execution environment. Environment return values are used to return values from user functions. If the architecture of the language were changed slightly, then return values could be received from any command block, and not just user functions. However, there should be no need to return values from other types of commands. To return values from user functions, the return statement must occur at the end of multiple command blocks or else the return value will be lost. Here is an example:

```
x10gimli test;
functions
   func1(inval) return inval + 3;
   func2(inval)
      begin
         return 5;
         inval = inval + 1;
      end
   func3(inval)
      begin
         begin
```

```
            return 3;
         end
         begin
            inval = 4;
         end
         begin
            return 6;
         end
      end
 start
    log (""+func1(5)+", "+func2(4)+", "+func3(2));
end x10gimli.
```

The output from this example will be:

```
"8, 5, 6"
```

The example demonstrates three potential attempts at returning values:

- In `func1` the return statement is performs as expected, and returns the summed value of the expression to the parent environment.
- The second function demonstrates that a return value can be set at any time during the execution block. The return value is set in the parent environment at the moment the `return` command is executed. The `begin` command propagates the return value to its parent environment.
- Finally, `func3` shows that the return value can be propagated through multiple levels. Returns can be propagated indefinitely, but are stopped at the termination of user functions. The second return is necessary because the command block that occurs after the first return causes the return value to be replaced by a null value.

Following the standard way to return values from functions, there should be no problem using this command.

**Usage:**

**return** *expression*;

## Motion Command

<*motion-command*> ::= **motion** <*variable*>**;** <*start-block*> <*finish-block*> | **motion** <*variable*> <*variable*>**;** <*start-block*> <*finish-block*>

This is a special command unique to X10Gimli. The motion response command essentially defines a temporary change of state, for a minimum amount of time. When the `motion` command is called, a start command is executed, and a new named trigger is created that will fire after the specified amount of time (60 seconds is the default if no time is specified). When the trigger fires, the finish command is executed. If another motion command is called on the same device or identifier before time has expired, it can be reset to fire after a new delay. This can be used easily for turning lights on and off, but can be applied to anything. For example:

```
begin
   define varx = 0;
   motion test1;
      start varx = 1;
      finish varx = 2;
   delay (5000);
   log(varx) ;
   motion test1 5;
   delay(6000);
   log(varx);
   motion A2 0:00:05;
end
```

The output of this example will be:

```
"1"
"2"
```

The example demonstrates three key pieces of the motion response command:

- Any identifier can be used to name the motion trigger. Individual start and finish commands can be defined. If either or both of the commands are omitted, then the default commands are used. The delay can be specified along with the motion trigger.
- It is possible to reset the motion trigger before it is fired. By calling the motion command again with the same identifier, a new time delay can be set. In this case, the time is less than what is actually remaining, but it can be more. Since only the trigger is being reset in this case, the start and finish commands do not need to be specified. The trigger already has the associated finish command.
- Special handling is done with X10 value identifier for the motion triggers. The default behavior of the X10 motion handler is to send an 'ON' command to the device with the specified address. The default behavior to finish it to send an 'OFF' command to the device with the identifier address. So, in the case given, the 'ON' command will be sent, and five seconds later, the 'OFF' command will be sent.

Being able to easily and powerfully implement timed motion detector responses is an integral part of this system.

### Usage:

**motion** *idvariable* {*timevariable*}; {**start** *startcommand*} {**finish** *finishcommand*}

# Trigger Command

<*trigger-command*> ::= **trigger** <*trigger*>

Local triggers can be created and reassigned with this command. Global triggers can also be reassigned with this command. Triggers that are created can be anonymous or named. Local triggers can only be fired once, and then they need to be redefined. The trigger definition follows the normal `trigger` syntax. Here is an example:

```
begin
   define currenttime = time;
   define varx = 0;
   define vary = 0;
   trigger (currenttime == (time - 5)) do varx = 1;
   trigger test1 (time == (time + 1)) do vary = 1;
   delay(1000) ;
   trigger test1 (varx == 1) do vary = 1;
   delay(5000);
   log (""+varx", "+vary);
end
```

The output from this code will be:

```
"1, 1"
```

The example shows a couple cases of handling triggers:

- Anonymous triggers can be created to perform tasks as defined. In the example, the trigger will fire after five seconds and execute its code.
- The named trigger is initially set to fire with a condition that will always be false. That same trigger is then reassigned a new condition and result. Since triggers are checked every second, as soon as `varx` receives its new value, the named trigger fires.

Remember that local triggers will only fire once and are then lost. Triggers are especially useful for timed tasks, but can be applied to any kind of condition.

**Usage:**

**trigger** {*triggeridentifier*} **(***triggerexpression***) do** *triggercommand*;

# Function Call

<*ident-command*> ::= <*variable*> = <*values*>**;** | <*variable*>**;**

This class handles execution of all user functions defined using X10Gimli source code. Parameters are passed by value and local definitions are created that correspond to the parameter names. So, within the function, access to all of the parameters is handled the same as with any other identifier. Functions can execute blocks of commands and return values. The following example illustrates some function call examples:

```
x10gimli test;
defintions
   defa = 1;
functions
   func1() begin define var1 = 1; end
   func2(defa) begin defa = 2; end
   func3(in1, in2) begin defa = 2; return in1 + in2; end
start
   begin
      func1();
      varz = 1;
      func2(varz);
      vary = func3(4, defa);
      log (""+var1+", "+varz+", "+vary+", "+defa);
   end
end x10gimli.
```

This example produces the following output:

```
"1, 1, 5, 2"
```

A few issues about he way user functions and parameter passing is done can be seen in the above example:

- Functions do not need parameters. A function with no parameters executes normally. It can return values and access all variables in scope.
- Parameters are passed by value and new definitions for those parameter values are created in the function's execution environment. In the example, `varz` is passed into `func2`. The parameter is fully evaluated before being passed, so `func2` creates a new definition in the local environment of `defa` = 1. In the function, the local `defa` definition is assigned a new value, so neither `varz` nor the global `defa` are affected.
- Multiple parameters can be passed in and work as previously described. In the example, `defa` gets assigned a new value, but the parameter `in2` was already assigned the evaluated `defa` value, so, the return value is based on that.

It is important to remember scope in dealing with user defined functions. Duplicate identifier names further in scope cannot be accessed. Parameters cannot be passed by reference.

### Usage:

*functionname*`(`*parameterlist*`);`

# Error Codes

**E100** `INVALID PARAM COUNT`
An invalid number of parameters was passed into a function.

**E101** `INCOMPATIBLE TYPE COMPARISON`
A comparison was attempted between two data types that are incompatible.

**E102** `INCOMPATIBLE TYPE ARITHMETIC`
An arithmetic operation was attempted between incompatible operands.

**E103** `INCOMPATIBLE TYPE PARAMETER`
The value passed into the function was of an incorrect type.

**E104** `NO VALUE`
There is no value to perform an operation on.

**E200** `INCORRECT SYNTAX`
Syntax error during parsing.

**E201** `INVALID TOKEN`
An illegal token was scanned during parsing.

# Language Components

The actual functionality of X10Gimli is provided by the external language components that allow system resources to be effectively used. The external language components are defined in Java classes that are automatically made available within the language. Without the external components, the language can not receive input, cannot send output, and is essentially dead. The available external language resources are:

- Functions
- Real-Time Identifiers
- Interface Gateways

Anyone can create new external components by following a set of conventions defined for each type of external system piece. As new components are created, the power of the language expands and its usefulness increases. A number of functions, identifiers and gateways are standard and already available.

This portion of the document contains short tutorials for creating new language components. The standard components are also listed and described later.

## Externally Defined User Functions Tutorial

Adding commands with more functionality than what can be achieved within the X10Gimli system can be done in Java. The FuncCommand class provides the necessary functionality to create a new user defined function to use in the language. The new functions can be created following a set of conventions for writing new functions that is described below.

For external user function classes to work in the language, they must:

- Follow the naming convention described later on.
- Descend from the `X10Gimli.Command.FuncCommand` class.
- Be created in the `X10Gimli.Command.Function` package.
- Overload the `execute` method of the `Command` class.
- Have a default constructor.

To assist with the implementation, the `FuncCommand` class has helper functions that should be used, and will be described in more detail later on. For consistent documentation of the usage of new functions, a JavaDoc convention can be used. This will all be described shortly. First, here is the source code used by the `changemode` function:

```
package X10Gimli.Command.Function;

import java.util.*;
import X10Gimli.Debug;
import X10Gimli.Token;
import X10Gimli.Command.*;
```

```
import X10Gimli.System.*;
import X10Gimli.Value.*;

/**
 * Changes the system's current mode to the one specified by the input
 * parameter.  If the mode name specified does not exist, then the
 * system returns to normal execution without any active mode.
 * <p>
 * Usage:
 * <p>
 * <blockquote><font size=5><code>
 * <b>changemode</b>(<i>mode</i>);
 * </code></font></blockquote>
 *
 * Parameters:
 * <p>
 * <blockquote>
 * <i>Ident</i> <b>mode</b> - the name of the mode to activate<br>
 * </blockquote>
 *
 * Example:
 * <p>
 * <blockquote><pre>
 *    changemode(NIGHTMODE);</pre>
 * </blockquote>
 */
public class ChangemodeCommand extends FuncCommand {
   /**
    * Executes this command as described in the class description.  Parameter
    * types and count must be correct.  Execution is done in the specified
    * environment.<p>
    *    @param env execution environement
    *    @return null
    */
   public ValueType execute(EnvironmentSmall env){
      super.execute(env);
      if (checkNumParams(1) && env != null){
         ValueIdent ident = getIdentParam(0);
         SystemModel m = env.getSystemModel();
         if (m != null){
            Mode mode = m.findMode(ident.getValueString());
            if (mode != null){
               m.changeMode(mode);
            } else {
               m.changeMode((Mode)null);
            }
         }
      }
      return null;
   }
}
```

Here are some steps that can be followed to create a user-defined function that is automatically accessible to the X10Gimli system.

1. Decide the name by which the function will be known in the language. The naming convention requires the class exist in the `X10Gimli.Command.Function` package. The name of the new class dictates the function identifier that the language uses. The class name is the function name (with only the first letter capitalized) followed by "Command". For example, if a function called "sprinkler" were desired in the language, its class name would be "SprinklerCommand".

2. Create the Java skeleton for the function. The following template can be used:

```
package X10Gimli.Command.Function;

import java.util.*;
import X10Gimli.Debug;
import X10Gimli.Command.*;
```

```
import X10Gimli.System.*;
import X10Gimli.Value.*;

/**
 * Description of the new function goes here.
 * <p>
 * Usage:
 * <p>
 * <blockquote><font size=5><code>
 * <b>newfunction</b>(<i>param1</i>, <i>param2</i>);
 * </code></font></blockquote>
 *
 * Parameters:
 * <p>
 * <blockquote>
 * <i>ParamType</i> <b>param1</b> - parameter description<br>
 * <i>ParamType</i> <b>param2</b> - parameter desripction<br>
 * </blockquote>
 *
 * Example:
 * <p>
 * <blockquote><pre>
 *    newfunction(p1, p2);</pre>
 * </blockquote>
 */
public class NewfunctionCommand extends FuncCommand {
   /**
    * Executes this command as described in the class description.  Parameter
    * types and count must be correct.  Execution is done in the specified
    * environment.<p>
    *    @param env execution environement
    *    @return null
    */
   public ValueType execute(EnvironmentSmall env){
      return null;
   }
}
```

The bolded text in this template should be replaced correctly. The JavaDoc comments should be written appropriately to represent the function being created. The class name should be changed to the desired name.

3. With the skeleton code prepared, the execute command needs to be written. The execute command can be written in any way desired, as long as it is understood how the system will react to it. Most functions should contain at least the following:

```
   public ValueType execute(EnvironmentSmall env){
      super.execute(env); // this prepares the input parameters to be used
      if (checkNumParams(??) && env != null){ // checks that the correct number of
                                              // parameters was passed in
         // body of function
      }
      return null;
   }
```

This should all be done to correctly prepare the parameters to be used. Further descriptions of the functions being used can be found in the description of the `FuncCommand` class in the JavaDoc.

4. To write the body of the function using the input parameters, it is good to understand how X10Gimli values work. Descriptions of the individual value classes and value classes in general can be found in the `X10Gimli.Value` package JavaDoc. Also, keep in mind that in the body of the new function, it is wise to use `FuncCommand` helper functions that retrieve parameters and doing type checking. This ensures that a certain level of error reporting is done.

5. If the function is supposed to return a value, then that can be done as well. To return a value, create a new instance of the type of value to be returned and set its data. Then return the value when the function ends.

That's really all that needs to be done to create a new function within the language. It can be called just like any other function in the language.

# User Functions

## ACTIVATE

Activates a switch within a specific control to begin handling input responses. The switch that is activated must be in scope.

**Usage:**

```
activate(control, switch);
```

**Parameters:**

*Ident* **control** - the identifier corresponds to the control ID
*List* **switch** - the list of values that allow a switch to be resolved.

**Example:**

```
activate(X10, {A5});
```

## CHANGEMODE

Changes the system's current mode to the one specified by the input parameter. If the mode name specified does not exist, then the system returns to normal execution without any active mode.

**Usage:**

```
changemode(mode);
```

**Parameters:**

*Ident* **mode** - the name of the mode to activate

**Example:**

```
changemode(NIGHTMODE);
```

## DEACTIVATE

Deactivates a switch within a specific control to begin handling input responses. The switch that is deactivated must be in scope. This is handy at time when certain input needs to be ignored.

Usage:

```
deactivate(control, switch);
```

**Parameters:**

*Ident* **control** - the identifier corresponds to the control ID
*List* **switch** - the list of values that allow a switch to be resolved.

**Example:**

```
deactivate(X10, {A5});
```

## DELAY

Suspends execution for a specified number of milliseconds. Actually sleeps the current thread. This allows the processor to be freed up during waiting loops.

**Usage:**

```
delay(milliseconds);
```

**Parameters:**

*Number* **milliseconds** - the amount of time to suspend execution

**Example:**

```
delay(1000);
```

## INTERFACE

Initializes and activates a specified interface and attaches it to the system model. If no appropriate interface can be found, or initialization of the interface is unsuccessful, then errors are posted and nothing is attached to the model. Some gateways require parameters, while others do not. Passing the parameter 'NULL' as the initialization parameter will cause no parameter to be sent to the gateway initialization function.

**Usage:**

```
interface(interfacename);
interface(interfacename, initparam);
interface(interfacename, initparam, name);
```

**Parameters:**

*Ident* **interfacename** - the name of the interface to be initialized and attached
*Value* **initparam** - the initialization parameter for the interface. This may be a list of parameters.
*Ident* **name** - the name that this interface will be known by

**Example:**

```
interface(WINMESSAGE);
interface(TCP, 4325);
interface(CM11A, "COM1", X10INPUT);
```

## LOG

Writes a user text message to the user log. This command is especially useful when some execution log output is desired, but the full execution log is not required.

**Usage:**

```
log(message);
```

**Parameters:**

*String* **message** - the user text message

**Example:**

```
log("This is a log message.");
```

## PACKET

Creates a packet value with the specified type, fields, and values. If the length of the tag and value lists is not equal, an error is posted. When creating the packet, the values of the tag list are all transformed into identifier values if not already so. The values within the value list are fully evaluated before being placed in the packet.

**Usage:**

```
packet(packettype, taglist, valuelist);
```

**Parameters:**

*Ident* **packettype** - the type of packet to be created
*List* **taglist** - the list of identifier tags for value referencing
*List* **valuelist** - the list of values stored in the packet

**Returns:**

*Packet* - the packet value that is constructed from the input parameters

**Example:**

```
pPacket = packet(PACK, {TAG1, TAG2}, {1, A});
```

## SENDPACKET

Sends a packet to all of the output gateways available from the system model. The packet can have a specific destination or not.

**Usage:**

```
sendpacket(destination, packet);
sendpacket(packet);
```

**Parameters:**

*String* **destination** - the destination of the packet
*Packet* **packet** - the actual packet data to send

**Example:**

```
sendpacket("GATEWAY2", packet(TEST, {}, {}));
sendpacket(packetval);
```

## SENDPACKETANDWAIT

Sends a packet to all of the output gateways available from the system model. The packet can have a specific destination or not. The function then waits for a response packet so a value can be returned. If no response is received within 10 seconds, a null value is returned.

**Usage:**

```
sendpacket(destination, packet);
sendpacket(packet);
```

**Parameters:**

*String* **destination** - the destination of the packet
*Packet* **packet** - the actual packet data to send

**Returns:**

*Value* - the value the comes in the response packet

**Example:**

```
sendpacketandwait("RESPONSEGATEWAY", packet(TEST, {}, {}));
sendpacketandwait(packetval);
```

## SOUND

Plays a sound file specified by the input string. The X10Gimli import path is searched for the first occurrence of the file.

**Usage:**

```
sound(filename);
```

**Parameters:**

*String* **filename** - the name of the sound file to play

**Example:**

```
sound("FX.WAV");
```

## TOCONTROL

Executes the system model's current switch in another control. This is done by refreshing the system's control identification list, while not touching the switch identification list. The new control/switch is executed in the same way it would be from outside.

**Usage:**

```
tocontrol(control);
```

**Parameters:**

*Ident* **control** - the identifier corresponds to the control ID

**Example:**

```
tocontrol(X10);
```

## TOCONTROLSWITCH

Executes a specific control/switch within the current mode. This is done by refreshing the system's control specification list, and also changing switch identification list. The new control/switch is executed in the same way it would be from outside.

**Usage:**

```
tocontrolswitch(control, switch);
```

**Parameters:**

*Ident* **control** - the identifier corresponds to the control ID
*List* **switch** - the list of values that allow a switch to be resolved.

**Example:**

```
tocontrolswitch(X10, {A, 1});
```

## TOGGLE

Toggles a switch within a specific control to begin handling input responses. The switch that is toggled must be in scope.

**Usage:**

```
toggle(control, switch);
```

**Parameters:**

*Ident* **control** - the identifier corresponds to the control ID
*List* **switch** - the list of values that allow a switch to be resolved.

**Example:**

```
toggle(X10, {A5});
```

## TOSWITCH

Executes another switch in the system model's current control. This is done by refreshing the system's switch identification list, while not touching the control specification list. The new control/switch is executed in the same way it would be from outside.

**Usage:**

```
toswitch(switch);
```

**Parameters:**

*List* **switch** - the list of values that allow a switch to be resolved.

**Example:**

```
toswitch({A, 10});
```

## X10ADDRESS

Sends out a specific X10 address command.

**Usage:**

```
x10address(address);
```

**Parameters:**

*X10* **address** - address to be sent out before function command

**Example:**

```
x10address(A5);
```

## X10COMMAND

Sends a specific X10 command to the specified house code with any extra data necessary. The `X10ADDRESS` command should be called before this to specify the device to receive the command.

**Usage:**

```
x10command(command, x10);
x10command(command, x10, num);
```

**Parameters:**

*Command* **command** - the X10 command to send
*X10* **x10** - the full address or housecode to receive the command
*Number* **num** - the extra data required for BRIGHT and DIM commands

**Example:**

```
x10command(ON, A5);
x10command(DIM, A5, 23);
```

## X10SWITCH

Uses the last X10 input received and sends its function and extra data to the switch specified by the input parameter.

**Usage:**

```
x10switch(switch);
```

**Parameters:**

*X10* **switch** - destination switch to receive X10 command

**Example:**

```
x10switch(A5);
```

# Real-time External Variables Tutorial

Adding specialized identifiers to the X10Gimli system that represent real-time changing data can be done in Java. The `ValueIdent` class provides the base functionality needed to create a new real-time identifier value to use in the language. The new identifiers can be created following a set of conventions for creating new pre-defined identifiers that is described below.

For real-time values to work in the language, they must:

- Follow the naming convention described later on.
- Descend from the `X10Gimli.Value.ValueIdent` package.
- Be created in the `X10Gimli.Value.PreDefined` package.
- Overload the `getValue` method of the `ValueType` class.
- Have a default constructor.
- Return a valid X10Gimli value.

The `Value` class provides all of the functionality that values must have, so implementation of the real-time value calculation is the only thing that needs to be written. For consistent documentation of the usage of new identifiers, a JavaDoc convention can be applied. Also, if any of the time related data types is the return type, then the `resetTime` method should be overloaded to return a time with the appropriate delay described in the *Trigger* description in the **Language Overview** portion of this document. This will all be described shortly. First, here is the source code used by the `time` real-time identifier:

```
package X10Gimli.Value.PreDefined;

import java.util.*;
import X10Gimli.Token;
import X10Gimli.Debug;
import X10Gimli.System.*;
import X10Gimli.Value.ValueIdent;
import X10Gimli.Value.Value;

/**
 * Represents the current time of day.
 * <p>
 * Usage:
 * <p>
 * <blockquote><font size=5><code>
 * <b>time</b>
 * </code></font></blockquote>
 *
 * Return Type:
 * <p>
 * <blockquote><code>
 * <i>Time</i>
 * </code></blockquote>
 */
public class ValueTime extends ValueIdent {
   /**
    * Returns the value as described in the class description.<p>
    *    @param env execution environement, likely does nothing
    *    @return the value this class represents
    */
   public Value getValue(EnvironmentSmall env){
```

```
      return new X10Gimli.Value.ValueTime(Calendar.getInstance());
   }

   /**
    * There is a one minute delay before a time trigger can fire again.<p>
    *    @param env environment does nothing
    *    @return one minute delay
    */
   public Calendar resetTime(EnvironmentSmall env){ Calendar temp =
Calendar.getInstance(); temp.add(Calendar.MINUTE, 1); return temp; }
}
```

Here are some steps that can be followed to create a real-time value that is automatically accessible within the X10Gimli system.

1. Decide the name by which the identifier that will represent the real-time value in the language. The naming convention requires that the class exist in the `X10Gimli.Value.Predefined` package. The name of the new class dictates the variable identifier that the language uses. The class name is the identifier name (with only the first letter capitalized) preceded by "Value". For example, if an identifier called "temperature" were desired in the language, its class name would be "ValueTemperature".

2. Create the Java skeleton for the identifier. The following template can be used:

```
package X10Gimli.Value.PreDefined;

import java.util.*;
import X10Gimli.Token;
import X10Gimli.Debug;
import X10Gimli.System.*;
import X10Gimli.Value.ValueIdent;
import X10Gimli.Value.Value;

/**
 * Identifier description should go here
 * <p>
 * Usage:
 * <p>
 * <blockquote><font size=5><code>
 * <b>identifiername</b>
 * </code></font></blockquote>
 *
 * Return Type:
 * <p>
 * <blockquote><code>
 * <i>ValueType</i>
 * </code></blockquote>
 */
public class ValueTime extends ValueIdent {
   /**
    * Returns the value as described in the class description.<p>
    *    @param env execution environement, likely does nothing
    *    @return the value this class represents
    */
   public Value getValue(EnvironmentSmall env){
      return new X10Gimli.Value.ValueNothing.value;
   }
}
```

   The bolded text in this template should be replaced correctly. The JavaDoc comments should be written appropriately to describe what the new identifier represents. The class name should be changed to the desired name.

3. With the skeleton code prepared, the `getValue` method needs to be written. The method can be written in any way desired, as long as it returns a non-null X10Gimli value of some kind.

4. If the method returns any of the time related data types, then the `resetTime` method should also be created and return the appropriate delay as was shown in the `time` identifier example above.

That's really all that needs to be done to create a new identifier that returns real-time information within the language. It can be used the same as any other variable in the language, except it cannot be assigned a value.

# Real-time External Variables

## ADDRESS

Represents the last X10 address input to be received into the system model.

**Usage:**

**`address`**

**Return Type:**

*`X10`*

## COMMAND

Represents the last X10 command input to be received into the system model.

**Usage:**

**`command`**

**Return Type:**

*`Command`*

## DATE

Represents today's date.

**Usage:**

```
date
```

**Return Type:**

*Date*

## DAY

Represents the current day of the week.

**Usage:**

```
day
```

**Return Type:**

*Day*

## DEVICE

Represents the device code of the last X10 address input to be received into the system model.

**Usage:**

```
device
```

**Return Type:**

*Number*

## INPUTPACKET

Represents the current input packet being handled by the system model.

**Usage:**

```
inputpacket
```

**Return Type:**

*Packet*

## MONTH

Represents the current month of the year.

**Usage:**

```
month
```

**Return Type:**

*Month*

## SOURCE

Represents the source of the current input packet in the system model.

**Usage:**

```
source
```

**Return Type:**

*String*

## TIME

Represents the current time of day.

**Usage:**

```
time
```

**Return Type:**

*Time*

## TYPE

Represents the current type of input packet being handled by the system model.

**Usage:**

```
type
```

**Return Type:**

*Ident*

# I/O Gateway Tutorial

Creating new gateways allows all types of input and output devices to communicate with the X10Gimli system. Three types of gateways can be created:

- Input
- Output
- Input and Output

Individual gateways can be initialized and linked to the environment using the `interface` command within the language. Steps for creating new gateways, and a code template is given below.

For new gateways to work in the language, they must:

- Follow the naming convention described later on.
- Descend from the `X10Gimli.Interface.InputGateway`, `X10Gimli.Interface.OutputGateway`, or `X10Gimli.Interface.InputOutputGateway` class.
- Be created in the `X10Gimli.Interface.Gateway` package.
- Utilize the `inputReceived` method for input gateways, and implement the `sendPacket` method for output gateways.
- Implement the `initialize` method of the `X10Gimli.Interface.BasicGateway` class.
- Have a default constructor.

Here are some steps that can be followed to create a new gateway that is automatically accessible to the X10Gimli system.

1. Decide the name by which the gateway will be known in the language. The naming convention requires the class exist in the `X10Gimli.Interface.Gateway` package. The name of the new class dictates the identifier that the language uses to access it. The class name is the gateway name (fully capitalized) followed by "Gateway". For example, if a gateway called "IRlink" were desired in the language, its class name would be "IRLINKGateway".

2. Create the Java skeleton for the function. The following template can be used:

```
package X10Gimli.Interface.Gateway;

import X10Gimli.Interface.*;
import X10Gimli.Debug;
import X10Gimli.System.Packet.*;
import X10Gimli.Value.*;

/**
 * <b>Input/Output</b> gateway and an explanation of the gateway here.
 *
 * <p>
 * Input:
 * <p>
 * <blockquote>
 *    Description of the input functionality of the gateway. <p>
 * </blockquote>
 *
 * Output:
 * <p>
 * <blockquote>
 *    Description of the output functionality of the gateway. <p>
 * </blockquote>
 *
 * Packet descriptions:
 * <p>
 * <blockquote>
 * <code><b>PACKETTYPE</b></code> - packet description
 *    <blockquote>
 *    <i>ValueType</i> <code><b>TAGNAME</b></code> - value description<br>
 *    <i>ValueType</i> <code><b>TAGNAME</b></code> - value description<br>
 *    </blockquote>
 * <code><b>PACKETTYPE</b></code> - packet description
 *    <blockquote>
 *    <i>ValueType</i> <code><b>TAGNAME</b></code> - value description<br>
 *    <i>ValueType</i> <code><b>TAGNAME</b></code> - value description<br>
 *    </blockquote>
 * </blockquote>
 *
 * Initialization Parameters:
 * <p>
 * <blockquote>
```

```
 *    <i>ValueType</i> <b>paramname</b> - parameter description<br>
 * </blockquote>
 */
public class GATEWAYGateway extends InputOutputGateway {
   /**
    * Initialization description.<p>
    *    @param paramname parameter descirption
    *    @return true if initialization is successful
    */
   public boolean initialize(Value paramname){
      // perform checks and initialization, along with debug output
      // return true if everything succeeds
      return false;
   }

   /**
    * Send packet description for output packets.<p>
    *    @param packet packet to transmit
    */
   protected void sendPacket(Packet packet){
   }
}
```

The bolded text in this template should be replace correctly. The Javadoc comments should be written appropriately to represent the gateway being created. The class name should be changed to the desired gateway name. The template should also be modified for the task of creating an input only or output only gateway.

3. With the skeleton code prepared, the initialize command needs to be written. The initialize command can written in any way desired, as long as it returns true when the initialization succeeds, so the system adds the gateway to the internal gateway lists.

4. For output gateways, the `sendPacket` command needs to be implemented to actually transmit the packet out the interface in the desired way. Programs desiring to use the gateway will call the `transmitPacket` method, which enqueues the packet to be send, and the output handler thread will pull the packet off the queue and use the `sendPacket` command to actually transmit it.

5. For input gateways, when input is received, the input packet needs to be stored by calling the `inputReceived` method, which will allow the packet to be distributed to listeners. Programs desiring to receive the input will implement the receivePacket method, which is called when enqueued input packets are being fired off to the listeners.

Following the outlined steps, a new gateway can be created that is immediately accessible to the language. Gateways are of critical importance in the operation of the system. By connecting the TCP gateway to the system, and using it on remote machines, a distributed home automation system can be attained, where all kinds of interface devices are used by the system to offer complete flexibility and control of the home.

# Gateways

## CM11A

**Input/Output** gateway that interfaces with a CM11A X10 serial device. Handles input and output of X10 data through a CM11A serial port interface. Input is handled for individual address and function packets. The CM11A is one of the most common X10 computer gateways. This implementation of the gateway requires the Java Communications package.

**Input:**

Receives input packets through the serial port, which come in the form of X10 addresses and functions.

**Output:**

Sends output packets through the serial port, which are transmitted over the power lines to control X10 devices.

**Packet descriptions:**

**`X10ADDRESS`** - stores X10 address data

*X10* **`ADDRESS`** - full X10 address

**`X10FUNCTION`** - stores X10 function data

*Command* **`COMMAND`** - X10 function
*X10* **`ADDRESS`** - X10 house code
*Number* **`DIMS`** - number of dims

**Initialization Parameters:**

*String* **portname** - name of the serial port that the CM11A is connected to

# JOY

**Input** gateway that receives state information from the joystick port. Currently button state information is contained in the input packet. Individual joysticks can be rewired and connected to other types of sensors to receive unique kinds of input. For example, a circuit of door and window sensors could interface with X10Gimli through the joystick gateway by wiring a button into the circuit.

**Input:**

Polls the state of the joystick every second and creates an input packet whenever the state changes.

**Packet descriptions:**

**`JOYSTICK`** - contains joystick state information

*Ident* **`BUTTON1`** - "UP" or "DOWN"
*Ident* **`BUTTON2`** - "UP" or "DOWN"
*Ident* **`BUTTON3`** - "UP" or "DOWN"
*Ident* **`BUTTON4`** - "UP" or "DOWN"

**Initialization Parameters:**

*Number* **joystickid** - number of the joystick to interface with

# MR26A

**Input** gateway that receives data from an MR26A X10 serial device. X10 address and function input is received, as well as media control functions from RF remote controls. This is often a preferred method of receiving X10 commands from remote controls because there is no delay receiving the input. That allows responses to be sent out with less lag than if input is received by way of the CM11A.

**Input:**

Receives input packets through the serial port, which come in the form of X10 addresses and functions, and media control packets. The media control commands are as follows:

"1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "AB", "ENTER", "DISPLAY", "SUBTITLE", "TITLE", "RETURN", "EXIT", "POWER", "RECALL", "SKIPUP", "SKIPDOWN", "VOLUP", "VOLDOWN", "MUTE", "PLAY", "STOP", "PAUSE", "RW", "FF", "RECORD", "RIGHT", "LEFT", "DOWN", "UP", "MENU"

**Packet descriptions:**

**`X10ADDRESS`** - stores X10 address data

*X10* **`ADDRESS`** - full X10 address

**`X10FUNCTION`** - stores X10 function data

*Command* **`COMMAND`** - X10 function
*X10* **`ADDRESS`** - X10 house code
*Number* **`DIMS`** - number of dims

**`MEDIACONTROL`** - stores media control information

*String* **`COMMAND`** - media control function
*Number* **`LEVEL`** - amount of function input

**Initialization Parameters:**

*String* **portname** - name of the serial port that the MR26A is connected to

# TCP

**Input/Output** gateway that handles TCP network communication. Implements the transmission and reception of X10Gimli packets between machines and processes. Connections can be named so some routing is done if source and destination information is available. The gateway can be activated as a full server/client, or just a client.

**Input:**

Receives input packets from connected sockets and passes them to the input listeners.

**Output:**

Sends output packets through the connected sockets. Some connections will have a name associated with them. In that case, a packet is only send to it if it is known to be a possible destination.

**Packet descriptions:**

This gateway relays packets and handles all packet types. One particular packet is a handled within the gateway and manages some connection issues. A description of this packet is as follows:

**`CONNECTION`** - network connection packet

*Ident* **COMMAND** - connection command
*Value* **EXTRA** - command parameter

**Initialization Parameters:**

*Number* **port** - listening port for accepting connections

## TTS

**Output** gateway that uses the Java Speech API to produce text-to-speech output. In this way, X10Gimli can provide voice feedback to input events, as well as provide system status, timed voice events, etc. A Java Speech API must be installed for this gateway to function.

**Output:**

Takes an output string and sends it the the Java Speech API.

**Packet descriptions:**

**TTS** - text-to-speech packet

*String* **TEXT** – text to be spoken

**Initialization Parameters:**

none

## UDP

**Input/Output** gateway that handles UDP network communication. Implements the transmission and reception of `X10Gimli` packets between machines and processes. The UDP gateway periodically polls output addresses to confirm that the output addresses are still valid. If an output address becomes invalid, then it is removed from the list.

**Input:**

Receives input packets using the listening socket, and passes them to the input listeners.

**Output:**

Sends output packets to all destination addresses in the list.

**Packet descriptions:**

This gateway relays packets and handles all packet types. One particular packet is a handled within the gateway and manages some connection issues. A description of this packet is as follows:

**`CONNECTION`** - network connection packet

*Ident* **`COMMAND`** - connection command
*Value* **`EXTRA`** - command parameter

**Initialization Parameters:**

*Number* **port** - port for receiving input packets

## WINAPI

**Input/Output** gateway that allows certain Windows API calls to be made. This allows X10Gimli to easily interact with other Windows processes. Usage of the WINAPI gateway requires a native library to handle to different functions. Some API calls will return values, so this is an input gateway for that reason.

**Input:**

The return values to certain API calls. The return values are received in response packets.

**Output:**

A Windows message is posted using the message number and two message parameters. Window destination is determined by two strings for the parent window, and two more for the child window.

**Packet descriptions:**

**`WINAPI`** - contains a Windows API call with associated parameters

*String* **`COMMAND`** - the API command to execute

"FINDWINDOW" - returns a *Number*

*String* **`WINDOWNAME`** - window name
*String* **`CLASSNAME`** - window class

"FINDCHILDWINDOW" - returns a *Number*

*String* **WINDOWNAME** - window name
*String* **CLASSNAME** - window class
*Number* **PARENT** - parent window to search from
*Number* **OLDEST** - first child to search

"SIMULATEKEYDOWN"

*Number* **KEY** - key to press

"SIMULATEKEYUP"

*Number* **KEY** - key to release

"POSTMESSAGE"

*Number* **HWND** - window handle
*Number* **MESSAGE** - message
*Number* **WPARAM** - first message parameter
*Number* **LPARAM** - second message parameter

"SETFOCUS"

*Number* **HWND** - window handle

"RESTOREFOCUS"

"CREATEPROCESS"

*String* **COMMANDLINE** – command line parameter

**Initialization Parameters:**

none

# INDEX

## W



## X

**X10GIMLI**
**X-10 General Interface Modal Language Idea**

**Documentation**
**and**
**Reference Guide**

Language Concept and Design by
Adam Lane