Object Generator User's Guide
javAPRSSrvr 4.3

Table of Contents

Section 1 - Introduction

ObjectGenerator provides a simple APRS object and item generator. As a client on javAPRSSrvr, multiple may exist in a single instance. The generator can monitor files for changes allowing on-the-fly changes without requiring a reboot. The generator can also be programmatically set allowing usage in applications like DStarMonitor.

This application operates as a client to javAPRSSrvr. It requires javAPRSSrvr 4.0 to provide the network interface and do the packet parsing.

Section 2 - Program Requirements and Description

ObjectGenerator is designed to run on any OS and Java VM capable of running javAPRSSrvr 4.0.

ObjectGenerator is comprised of a number of classes which Java looks at as objects. ObjectGenerator classes are in ObjectGenerator.jar which must be specified in the ClassPath property in the ObjectGenerator's properties file.

ObjectGenerator can function in one of two basic modes: file or API. In file mode, ObjectGenerator will monitor a specified configuration file for updates. This configuration file is a Properties file which contains an object or item name (review the appendix on Properties file formatting for more information on using special characters in a property name) and the object's Properties file name as the value. Adding or deleting an object from this file is how an object can be dynamically started or killed while javAPRSSrvr is running. Modifying the individual object Properties files allow dynamic modification of an object/item while it is still being transmitted.

The API mode allows another client or application to add, delete, and modify objects and items without having to update files or stop and start javAPRSSrvr. This mode is used by DStarMonitor to generate the D-STAR repeater objects.

Object names and Item names must be unique and if you are using file mode, object/item properties files must be unique.

Section 3 - Configuration Properties

The configuration properties reside in properties files for each client adjunct, server adjunct, and port. The main properties file is called javaprssrvr.properties by default. You can use any text file for the main properties file if you pass the name into javAPRSSrvr as a command line parameter.

The property names are not case sensitive but the values can be. Defaults are shown below.

NOTE: UNLESS YOU REQUIRE A SETTING OTHER THAN THE DEFAULT, DO NOT INCLUDE ANY PARAMETERS WITH DEFAULT SETTINGS.

List parameters (L) may be defined on the property line or may be defined in a text file with the suffix .lst. If defined on the line, each entry is separated by a semicolon. If defined in a file, each entry is put on a separate line in the .lst file and the file name is the property value. Do not put blank lines in the file. For instance, this could be a definition for ListProperty (example only):

ListProperty=first.aprs.net:1313;second.aprs.net:1313

Or you could have the following 2 lines in a file named hubs.lst:

first.aprs.net:1313
second.aprs.net:1313

with ListProperty=hubs.lst

Properties preceded by a (M) are unchangeable and should not be included in your properties files. They are included in the descriptions below to indicate what common properties are available vs. those that have been forcibly overridden.

javAPRSSrvr Properties

Clients=

This must include the file of this ObjectGenerator’s properties file.

ObjectGenerator Properties

ClassPath=

(L) Must include ObjectGenerator.jar.

Class=

Must be set to net.ae5pl.objgen.ObjectGenerator.

StationCall=

This is the callsign-SSID for the ObjectGenerator.
It must conform to AX.25 standards and it must be different from javAPRSSrvr's ServerCall (the server's callsign-SSID) and any other station’s callsign-SSID visible to APRS-IS.

ObjectFile=

This points to the Properties file which lists each object/item and its respective definition file. Each property in the ObjectFile is an Object or Item name and the property value is the filename of the properties file for that object. For example, the entry MYOBJECT=spclobject.properties says the Object or Item definition contained in spclobject.properties will use the name MYOBJECT. Object/Item names can consist of up to 9 “printable ASCII” characters. See information at end for including special characters in a Property name. The path in ObjectFile will be used to resolve the location of each object/item’s properties file. For instance, if ObjectFile=objects/objects.properties then each object/item properties file will be search for in the objects subdirectory under javAPRSSrvr’s directory.

SendInterval=20

Default minutes to wait between transmissions.

Delay=30

Time (seconds) to wait before first transmission and interval to check ObjectFile for further updates.

(M)Upstream=false

(M)FullFeed=false

(M)ReadOnly=false

(M)LocalOnly=false

(M)History=false

(M)ServerAdjuncts=false

(M)LastHeardTime=-1

(M)MessageHoldTime=-1

Object/Item Properties

Latitude=

This specifies the latitude of the Object/Item.
The latitude is specified in decimal degrees, south is negative.

Longitude=

This specifies the longitude of the Object/Item.
The longitude is specified in decimal degrees, west is negative.

Compressed=false

This specifies if posits are to use compressed format.

Ambiguity=-1

This specifies how many digits of are considered valid (-1 = all).

Altitude=

Altitude in feet MSL of station.

Symbol=

This specifies the APRS symbol to use in the object/item.

DataExtension=

This is the 7 byte extension for PHG, etc.

Comment=

This specifies text to follow the posit data.

TimeStamp=

If this is set, it must conform to the APRS specification (for instance MDHz or HMSh, etc.). If this is not set, the entity is considered an Item, not an Object and is transmitted that way. If set, it is not validated other than being 6 digits followed by either a ‘z’, ‘/’, or ‘h’ and the entity is considered an Object and transmitted as such.

SendInterval=default SendInterval interval

This specifies the position beacon rate in minutes.

Delay=default Delay delay

This specifies how long to initially wait before beaconing in seconds.

Expires=

If set, this determines when the object will be killed (won’t be started if Expires is before object file is recognized). The format is using the java.util.Date.parse(String) mechanism and the time zone is defaulted to UTC. If you use local time, use an approved time zone designator of either an offset from UTC or a designator such as CDT for Central Daylight Time.

Section 4 - Installation Instructions

ObjectGenerator.jar must be placed in the wherever specified in the ClassPath property. Normally it will be in the same directory as javAPRSSrvr and ClassPath=ObjectGenerator.jar Add the properties file to Clients and start javAPRSSrvr. If using file mode, you will need a file containing a property for each Object or Item and related properties files for each defined Object and Item.

Section 5 – XML Status Page Description

General XML

<clientrcv>
<time>
<connect utc="1341672397366"/>
<lastlinein utc="1341685637381"/>
</time>
<upstream>
false
</upstream>
<readonly>
false
</readonly>
<login>
<callssid verified="true">
AE5PL-OG
</callssid>
<software version="4.0.0">
ObjectGenerator
</software>
</login>
<rcvdfrom bytes="5444" lines="58" packets="58">
<udp bytes="0" lines="0" packets="0"/>
</rcvdfrom>
</clientrcv>

Detail XML


```
<clientrcv>
<time>
<connect utc="1341672397366"/>
<lastlinein utc="1341686237381"/>
</time>
<class name="ObjectGenerator">
<package name="net.ae5pl.objgen" revision="b01" title="APRS Object Generator" version="4.0.0"/>
</class>
<messages callssids="0" unacked="0"/>
<upstream>
false
</upstream>
<readonly>
false
</readonly>
<login>
<callssid verified="true">
AE5PL-OG
</callssid>
<software version="4.0.0">
ObjectGenerator
</software>
</login>
<rcvdfrom bytes="5628" lines="60" packets="60">
<udp bytes="0" lines="0" packets="0"/>
</rcvdfrom>
<objects total="3">
<object>
<![CDATA[W5MRC-RW ]]>
</object>
<object>
<![CDATA[W5MRA-R ]]>
</object>
<object>
<![CDATA[MARCMTG ]]>
</object>
</objects>
</clientrcv>
```

Section 6 - Property file format

From java.util.Properties Javadoc:

Properties are processed in terms of lines. There are two kinds of line, *natural lines* and *logical lines*. A natural line is defined as a line of characters that is terminated either by a set of line terminator characters (\n or \r or \r\n) or by the end of the stream. A natural line may be either a blank line, a comment line, or hold all or some of a key-element pair. A logical line holds all the data of a key-element pair, which may be spread out across several adjacent natural lines by escaping the line terminator sequence with a backslash character \. Note that a comment line cannot be extended in this manner; every natural line that is a comment must have its own comment indicator, as described below. Lines are read from input until the end of the stream is reached.

A natural line that contains only white space characters is considered blank and is ignored. A comment line has an ASCII '#' or '!' as its first non-white space character; comment lines are also ignored and do not encode key-element information. In addition to line terminators, this format considers the characters space (' ', '\u0020'), tab ('\t', '\u0009'), and form feed ('\f', '\u000C') to be white space.

If a logical line is spread across several natural lines, the backslash escaping the line terminator sequence, the line terminator sequence, and any white space at the start of the following line have no affect on the key or element values. The remainder of the discussion of key and element parsing (when loading) will assume all the characters constituting the key and element appear on a single natural line after line continuation characters have been removed. Note that it is *not* sufficient to only examine the character preceding a line terminator sequence to decide if the line terminator is escaped; there must be an odd number of contiguous backslashes for the line terminator to be escaped. Since the input is processed from left to right, a non-zero even number of 2*n* contiguous backslashes before a line terminator (or elsewhere) encodes *n* backslashes after escape processing.

The key contains all of the characters in the line starting with the first non-white space character and up to, but not including, the first unescaped '=', ':', or white space character other than a line terminator. All of these key termination characters may be included in the key by escaping them with a preceding backslash character; for example,

\:\=

would be the two-character key ":=". Line terminator characters can be included using \r and \n escape sequences. Any white space after the key is skipped; if the first non-white space character after the key is '=' or ':', then it is ignored and any white space characters after it are also skipped. All remaining characters on the line become part of the associated element string; if there are no remaining characters, the element is the empty string "". Once the raw character sequences constituting the key and element are identified, escape processing is performed as described above.

As an example, each of the following three lines specifies the key "Truth" and the associated element value "Beauty":

```
 Truth = Beauty
  Truth:Beauty
 Truth                :Beauty
```

As another example, the following three lines specify a single property:

```
fruits                  apple, banana, pear, \
                        cantaloupe, watermelon, \
                        kiwi, mango
```

The key is "fruits" and the associated element is:

"apple, banana, pear, cantaloupe, watermelon, kiwi, mango"

Note that a space appears before each \ so that a space will appear after each comma in the final result; the \, line terminator, and leading white space on the continuation line are merely discarded and are *not* replaced by one or more other characters.

As a third example, the line:

```
cheeses
```

specifies that the key is "cheeses" and the associated element is the empty string "".

Characters in keys and elements can be represented in escape sequences similar to those used for character and string literals (see sections 3.3 and 3.10.6 of *The Java™ Language Specification*). The differences from the character escape sequences and Unicode escapes used for characters and strings are:

- Octal escapes are not recognized.
- The character sequence \b does *not* represent a backspace character.
- The method does not treat a backslash character, \, before a non-valid escape character as an error; the backslash is silently dropped. For example, in a Java string the sequence "\z" would cause a compile time error. In contrast, this method silently drops the backslash. Therefore, this method treats the two character sequence "\b" as equivalent to the single character 'b'.
- Escapes are not necessary for single and double quotes; however, by the rule above, single and double quote characters preceded by a backslash still yield single and double quote characters, respectively.
- Only a single 'u' character is allowed in a Unicode escape sequence.