XML Tagging Guidelines for 1-Wire Devices

Revision 1.00
September 1, 2001

by Brian D. Hindman

Table of Contents

Abstract

All Dallas Semiconductor 1-Wire devices, including iButtons, are individually assigned a 64-bit 1-Wire Network Address number. Each number is laser engraved into the read-only memory of each device. Dallas Semiconductor manages this number pool of 10^{19} entries so that each device has a guaranteed unique number assigned to it.

Once these 1-Wire devices leave Dallas Semiconductor, most customers will associate the 1-Wire Network Address number with a physical object. The number can then be placed in a database and the physical object tracked. With the introduction of more complex 1-Wire devices that perform sophisticated sensing, instead of just tracking the object, the object can now be analyzed or even manipulated as well. This, combined with the desire to group 1-Wire devices together into a cluster to perform a group function, makes a 1-Wire Tagging scheme desirable.

This document will present a *1-Wire TAG* format that describes the aforementioned associations, groupings, and sensing operations. The *1-Wire TAG* can be thought of as data that can reside in a traditional database, a file on a hard drive, or even in the memory of a 1-Wire device. The data indicates the purposes of the 1-Wire device(s), their locations, and specific software classes to service and control them. By carrying the *1-Wire TAG* with a cluster of 1-Wire devices, the cluster can be self-describing and self-configuring when presented to a new master application, such as a web site or a software program.

Tagging specifications for 1-Wire devices are already in existence. Much of the work contained herein has been pulled from two different projects. The first is “Tagging Guidelines for 1-Wire Sensors and Instruments” which can be found here: ftp://ftp.dalsemi.com/pub/auto_id/softdev/softdev.html (look under “New TMEX examples” and then the link for “1-Wire Tagging”). This project describes a tagging format that is very powerful with script-like features. It has the advantage of producing very small tag files which can be handled by 1-Wire devices with small memory footprints, but can be a bit complex to implement. The second project is “XML Java 1-Wire Tagging” and can be found here: http://xml1-wire.sourceforge.net/. The advantage of this format is the use of XML (eXtensible Markup Language) to make tags easier to read or “parse”, not only by humans, but also by a vast array of computer/internet software. Although the resulting tags are much larger, its strengths lie in the ease of integration and extensibility. The project described in this document also uses XML, but implements the design similar to the non-XML version.

1-Wire TAG Specification

The *1-Wire TAG* file specification as defined here can reside in a file on the 1-Wire sensor itself (physical) or in another file system such as on the master (local) or in a database on a server (remote). When the file is located in the memory of a 1-Wire device in the 1-Wire Cluster (physical) it must be in the form of an *Extended File Structure* file. The file can reside in any normal memory 1-Wire device such as the DS1993, DS2406, or DS2433. The *1-Wire File Structure* is the format used by the TMEX software drivers. The specification for the *1-Wire File Structure* can be found in the Dallas Semiconductor Application Note 114, "1-Wire File Structure".

The tag file contains data represented in XML that describe the 1-Wire device's characteristics and operations and/or an entire cluster or clusters of 1-Wire devices. The contents of the tag file is parsed by the master application to obtain the available characteristics and operations of the device or cluster and provides self-registering capabilities to even unknown configurations of sensor clusters. The file should preferably be named TAGX.000 on a 1-Wire device.

1-Wire XML Tag Definition

A 1-Wire XML Tag is a valid XML data object (can be thought of as a file or data stream) that contains pre-defined XML elements and attributes that describe a 1-Wire Device or cluster of 1-Wire devices. The XML data object then is read and parsed by a master application (such as a Java^{TM} program) and software objects are then created to read each 1-Wire device, manipulate it, and show its various relations to other devices or locations on the 1-Wire network.

Design Objectives

This is a small list of design objectives that went into the making of this specification.

1) When designing the XML tag, make it as small as possible while maintaining human readability.
    a. By keeping the XML tag elements small in size.
    b. When writing tag files into 1-Wire devices, remove all unnecessary white space.
    c. By using empty XML elements, called mini tags, in small memory situations.
2) The master application, which, for the purposes of this specification are written in Java, should be able to parse multiple XML tags to describe an entire 1-Wire network. Multiple tags can reside either locally, remotely, or on the 1-Wire devices themselves.
3) When parsing the XML tags, the master application should create a list of Java objects that can fully describe, read, and manipulate each tagged 1-Wire device. This list should be kept as "flat", simple, and small as possible.
4) For communicating to the 1-Wire devices, use the 1-Wire API for Java, which can be found here: http://www.ibutton.com/software/1wire/1wire_api.html. As part of the Java objects, make use of the powerful OneWireContainer classes and make sure each object contains a specific OneWireContainer for the particular tagged device.
5) Include support for 1-Wire branching.
6) Parse the XML tags with an event-based parser (SAX) which will enable small devices (including TINI and handhelds) to use the 1-Wire XML tagging format

described in this document. Since, at the time of this writing, it has been verified that at least two SAX-based XML parsers run successfully on TINI, they should be used to verify the design. The first parser is MinML and can be found here: http://www.minml.com. This was designed specifically with TINI in mind. The second is NanoXML, which can be found here: http://nanoxml.sourceforge.net.

Parent Elements

There are four parent 1-Wire XML elements that make up an XML tag file. A 1-Wire device can be classified in this specification as one of three different objects (known collectively as *tagged devices*). These make up the first three parent XML elements: *branch*, *sensor*, and *actuator*. The fourth element is a *cluster* and represents a specific group of tagged devices. Please keep in mind that XML is case sensitive in nature and that these elements should be in lower case.

Branch

A *branch* tagged device represents Dallas Semiconductor's line of 1-Wire switches (DS2406, DS2409, etc.). Thus, a 1-Wire network could be thought of as a tree, and to read or manipulate a particular tagged device, one might have to go through a few branches to do it. The following is an example branch XML element:

```
<branch addr="77000000023CEC12">
        <label>Weather Station Switch</label>
        <channel>1</channel>
        <init>0</init>
                .
                .
                .
        (Other sensor, actuators, or branches can go here)
                .
                .
                .
</branch>
```

The XML element, <branch>, consists of a single attribute *addr*, along with at least three child elements, <label>, <channel>, and <init>. The attribute value of *addr* is the tagged device's 1-Wire Net Address. The child element, <label>, is a text description of the branch, the second child element, <channel>, is a number representing which switch on the tagged device to close, and <init> is a number representing the initial state of the switch.

Other child elements are possible. These can be other <sensor>, <actuator>, or <branch> elements. Thus, branches can be nested.

Sensor

A *sensor* tagged device represents any 1-Wire device that can be used to “read” or “sense” something. It could be something as simple as detecting a particular 1-Wire device’s presence on the 1-Wire bus, to reading a temperature from one of the many different thermometer devices available. The following is an example XML sensor element:

```
<sensor addr="DD0000020057A101" type="Contact">
        <label>Northeast</label>
        <max>Making contact</max>
        <min>No contact</min>
</sensor>
```

The XML element, <sensor>, consists of two attributes, *addr* and *type*, along with the child elements that are deemed necessary for a particular type. The attribute value of *addr* is the tagged device’s 1-Wire Net Address. The second attribute, *type*, is the type of sensor. When parsing the XML file, this attribute will determine what kind of software object gets created for a particular type. In this example, the sensor is of type “Contact”, and according to the type “Contact” (please see the section of this document entitled “Types”) the following three child elements are needed: <label>, <max> and <min>.

The child element, <label>, is a text description of the sensor. The child element, <max>, is a text message describing what occurs when the tagged device is in a certain state. In this instance, <max> represents when the tagged device is present on the 1-Wire bus. The child element, <min>, is similar except that <min> (usually being the opposite of <max>) is a message describing the state of the tagged device when it is not connected to the 1-Wire bus.

Actuator

An *actuator* represents any tagged device that can be acted upon or exercised. This could be something as simple as flipping a switch and making a buzzer sound to setting the wiper value of a 1-Wire potentiometer and adjusting the light intensity in a room. The following is an example XML actuator element:

```
<actuator addr="B700000018AE3212" type="Switch">
          <label>Buzzer</label>
          <max>Buzz!!!</max>
          <min>Sleep</min>
          <channel>0</channel>
          <init>0</init>
</actuator>
```

The XML element, <actuator>, consists of two attributes, *addr* and *type*, along with the child elements that are deemed necessary for a particular type. The attribute value of *addr* is the tagged device's 1-Wire Net Address. The second attribute, *type*, is the type of actuator. In this case, the type is "Switch".

Different actuators may have different numbers of child elements. For the particular actuator of type "Switch", these specific child elements are needed: <label>, <max>, <min>, <channel>, <init>. The type attribute is important. When parsing the XML file, this attribute will determine what kind of software object gets created for a particular tagged device. For a list of sensor and actuator types, please see the section of this document entitled "Types".

The child element, <label>, is a text description of the actuator. The child element, <max>, is a text field representing a choice or a selection of a state into which the actuator can be placed. It can be thought of as an item in a drop-down menu of a software application. In this instance, <max> represents the choice of connecting the switch to make the buzzer sound. The child element, <min>, is similar, except that <min> usually represents the opposite choice which, in this case, is switching the buzzer to the off or "Silent" position.

Cluster

A *cluster* is a grouping of tagged devices and/or other clusters. It is made up of a tag and a single XML attribute, "name". It can also have child elements made up of any other 1-Wire parent element (branch, sensor, actuator, or cluster). Therefore, *cluster* is an element that can be nested. The following is an example XML cluster element:

<cluster name="Weather Station">
.
. (Other elements, such as *branch*, *sensor*, *actuator*, or even other *clusters* would go here).
.
</cluster>

Child Elements

Child elements are used only by the parent elements of tagged devices (<branch>, <sensor>, and <actuator>). The six child elements are as follows: <label>, <max>, <min>, <channel>, <init>, and <scale>. In this specification, all child elements can belong to any tagged device. When parsed into software objects, each software object (representing a tagged device) should contain a data member for each child element. However, depending upon the type attribute of the tagged device, some elements will not be used. The following is an example of this:

Example 1:

```
<sensor addr="490000000212D016" type="Contact">
        <label>Employee 22 badge</label>
        <max>Making contact</max>
        <min>No contact</min>
 </sensor>
```

Example 2:

```
<actuator addr="B700000018AE3212" type="Switch">
        <label>Buzzer</label>
        <max>Buzz!!!</max>
        <min>Sleep</min>
        <channel>0</channel>
        <init>0</init>
</actuator>
```

Example 1 above shows a sensor of type “Contact”. It uses the <label>, <max>, and <min> child elements. However, when parsed, the software object created for this particular tagged device will contain all six child elements as data members. Example 2 shows an actuator of type “Switch” and uses five of the six child elements. Again, the software object created for “Switch” tagged devices will contain all six child elements as data members, but only five of them will be used.

Keep in mind that child elements do not necessarily mean the same thing for each “type”of tagged device. Although the element <label> has a generic meaning across all types, most of the other elements do not. The above two examples show <max> and <min> child elements with different meanings. Example 1 uses <max> and <min> to describe the two different possible states of the “Contact” tagged device. Example 2 shows <max> and <min> being used as state selections. This means that they are used to select the state into which the tagged device can be placed. And, finally, Example 3 shows <max> being used by itself as an event message.

Example 3:

```
<sensor addr="B200000018BC2A12" type="Event">
        <label>Switch #1</label>
        <max>Activity sensed!</max>
        <channel>1</channel>
</sensor>
```

Table 1. Child Elements Used by Tagged Type

(The “x” marks a “used” element.) This table shows a list of types and the child elements used by each type.

	label	*max*	*min*	*channel*	*init*	*scale*
Contact	x	x	x			
Event	x	x		x		
Switch	x	x	x	x	x	
Thermal	x					
Humidity	x					
D2A	x			x		x

Label

The <label> child element is used primarily to describe the tagged device. It can be used to label a device, give location information, and give revision numbers or dates. Here is an example:

```
<sensor addr="E200000006283826" type="RH10">
        <label>Indoor Humidity Sensor</label>
</sensor>
```

In the above example, the <label> element describes the “RH10” sensor as an “Indoor Humidity Sensor”.

Max

The <max> child element can be used in several different ways. This document shows the use of <max> in three specific ways. They are described above with three helpful examples under the section “Child 1-Wire XML Elements”. Keep in mind that <max> nor any other child element is limited to the definitions contained in this document. It is left to the individual developer to give meaning to the child element when creating a new “type” of sensor or actuator.

Min

Similar to <max>, the <min> child element can be used in different ways. In the “Contact” type of tagged device (see the “Types” section of this document), <min> is used to describe one of the two different possible states that the device can have. The “Switch” type of tagged device shows <min> being used as a state selection. And, finally, although no type of tagged device implements it, <min> can be used by itself as an event message similar to <max> in the discussion above.

Channel

The <channel> child element is generally used to select a particular choice from an array of choices. It should also represent an integer number. For example, in the types "Branch", "Switch", and "Event", the child element <channel> is used to choose a particular switch from an array of 2 switches. Please see Example 1 below:

Example 1:

```
<sensor addr="B200000018BC2A12" type="Event">
        <label>Switch #1</label>
        <max>Activity sensed!</max>
        <channel>1</channel>
</sensor>
```

In the type, "D2A", <channel> is used to choose a particular digital potentiometer to exercise from a possible array of many. In Example 2 below, channel is "0" (since, at the time of this writing no 1-Wire device contains multiple potentiometers).

Example 2:

```
<actuator addr="BB0000000062602C" type="D2A">
        <label>Red Light</label>
        <channel>0</channel>
        <scale>Intensity</scale>
</actuator>
```

Init

The <init> child element is generally used as a state initializer for tagged devices, especially actuators. If there is any state that the tagged device should be in before it is exercised, this is the child element to use. It can be any data type needed. In Example1 below, the tagged device of type "Switch" uses the child element <init> to initialize the "Switch" to the "off" position (represented here by the number 0).

Example 1:

```
<actuator addr="B700000018AE3212" type="Switch">
        <label>Buzzer</label>
        <max>Buzz!!!</max>
        <min>Sleep</min>
        <channel>0</channel>
        <init>0</init>
</actuator>
```

Scale

The <scale> child element is generally used for tagged devices (mostly actuators) when a scale is needed to display results or display the kinds of state selections available. In Example 1 below, the tagged device of type "D2A" uses the child element <scale> to represent a volume scale in decibels.

Example 1:
```
<actuator addr="BB0000000062602C" type="D2A">
        <label>Stereo System</label>
        <channel>0</channel>
        <scale>Volume in decibels</scale>
</actuator>
```

Types

Types are very important to 1-Wire Tagging. Each tagged device (whether a sensor, actuator, or branch) will have a unique software object created for it based on its type. Although the software object will contain all the child elements listed above as data members, the implementations of the methods will be different. This will be based on the type attribute parsed for a tagged device. At the time of this writing, six types have been identified and implemented: Contact, Event, Switch, Thermal, RH10, and D2A. Please note that the value of type should always be capitalized in a 1-Wire XML file. In the Java implementation of the XML 1-Wire tagging scheme, the type is the actual *name* of the software object, and it is instantiated dynamically at run-time.

Contact

A tagged device of type “Contact” is a sensor because, in essence, it “senses” or “reads” something. Here its purpose is to sense if the tagged device is present on the 1-Wire bus. A “Contact” sensor has three child elements: <label>, <max>, and <min>. Of course, <label> is used to describe the sensor. In the example below, <label> indicates that this 1-Wire device is the badge of employee number 22. The <max> child element describes the state of the device being present on the 1-Wire bus, and the <min> child element describes the opposite state. Notice that the type, “Contact”, is capitalized.

Contact Example XML tag:
```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="490000000212D016" type="Contact">
        <label>Employee 22 badge</label>
        <max>Making contact</max>
        <min>No contact</min>
 </sensor>
```

Event

A tagged device of type “Event” is a sensor. Its purpose is to sense if activity has occurred on a particular 1-Wire switch. An “Event” sensor has three child elements: <label>, <max>, and <channel>. Of course, <label> is used to describe the sensor. In the example below, <label> specifies that the indicated 1-Wire device is “Switch #1” (of possibly many other switches) on the 1-Wire bus. The <max> child element is the event message whenever activity on the switch has been sensed. And, the <channel> child element represents the specific switch on the 1-Wire tagged device to be sensed.

Event Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="B200000018BC2A12" type="Event">
        <label>Switch #1</label>
        <max>Activity sensed!</max>
        <channel>1</channel>
</sensor>
```

Switch

A tagged device of type "Switch" is an actuator. Here, its purpose is to exercise a 1-Wire switch. A "Switch" actuator has five child elements: <label>, <max>, <min>, <channel>, and <init>. In the example below, <label> is used to describe the actuator as a buzzer. The <max> child element describes a state selection. If the <max> state is chosen, the "Buzzer" will "Buzz!!!". The <min> child element also describes a state selection, but, of course, it is opposite to <max>. If the <min> selection is chosen, the buzzer will stop buzzing and "Sleep". The <channel> child element represents the specific switch on the 1-Wire tagged device to be exercised, and the <init> child element describes the initial state of the switch which, in this case, is disconnected (represented by the integer 0).

Switch Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<actuator addr="B700000018AE3212" type="Switch">
        <label>Buzzer</label>
        <max>Buzz!!!</max>
        <min>Sleep</min>
        <channel>0</channel>
        <init>0</init>
</actuator>
```

Thermal

A tagged device of type "Thermal" is a sensor because its purpose is to sense the temperature of its surroundings. A "Thermal" sensor has only one child element, <label>. In the example below, <label> indicates that this tagged device is sensing an indoor temperature. Nothing more is needed as the software object created through parsing the XML tag file will produce a OneWireContainer as part of the "Thermal" software object. The software object implements the TemperatureContainer interface. Please see the documentation for the 1-Wire API for Java located here: http://www.ibutton.com/software/1wire/1wire_api.html.

Thermal Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="E200000006283826" type="Thermal">
        <label>Indoor Temperature</label>
</sensor>
```

Humidity

A tagged device of type “Humidity” is a sensor because its purpose is to sense the humidity of its environment. A “Humidity” sensor has only one child element, <label>. In the example below, <label> indicates that the specified tagged device is sensing an indoor humidity. Nothing more is needed as the software object created through parsing the XML file will produce a OneWireContainer as part of the “Humidity” software object. This specific OneWireContainer implements the HumidityContainer and ADContainer interfaces. Please see the documentation for the 1-Wire API for Java located here: http://www.ibutton.com/software/1wire/1wire_api.html.

Humidity Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="E200000006283826" type="Humidity">
        <label>Indoor Humidity</label>
</sensor>
```

D2A

A tagged device of type “D2A” is an actuator because its purpose is to exercise or “actuate” a digital-to-analog device. In this case, the device is a 1-Wire digital potentiometer. A “D2A” actuator has three child elements: <label>, <channel>, and <scale>. In the example below, <label> is used to describe the actuator as a dimming red fluorescent light. Since it is an actuator, it will have state selections. However, the XML file does not need to contain these. The only “D2A” tagged device at the time of this writing is the DS2890 digital potentiometer, and the software object created as a result of parsing the XML file below will know of its state selections through its OneWireContainer which implements PotentiometerContainer. The PotentiometerContainer can be used to determine that the DS2890 has 256 wiper states. Finally, the <channel> child element represents the specific potentiometer on the 1-Wire tagged device to be exercised (in this case, there is only one).

D2A Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<actuator addr="BB0000000062602C" type="D2A">
        <label>Dimming red fluorescent light</label>
        <channel>0</channel>
        <scale>Light intensity</scale>
</actuator>
```

Mini Tags

Due to memory constraints on 1-Wire devices, it is highly desirable to make 1-Wire XML tags as small as possible. One way that this can be accomplished is through mini tags. In short, mini tags are single XML elements with empty content. They reside in a raw ASCII form in the memory of the 1-Wire device. In this specification, they should be ignored if found outside of 1-Wire devices.

Mini Tag Format

A mini tag has the format of <TTNNCI/> in standard ASCII. TT is a 2-letter identifier for the "Type" of tagged device. NN is a 2-digit identifier signifying the implementation number of a device. "C" is optional and is a number corresponding to the <channel> child element tag. "I" is also optional and is a number corresponding to the <init> child element tag. Please see the sections pertaining to <channel> and <init> in this document under "Child 1-Wire XML Elements". In the unlikely event that an "I" identifier exists without a "C" identifier, underscores should be placed where the "C" digit would normally go.

The empty XML element, <HU10/>, if it is found in the memory of a 1-Wire device, can be considered a mini tag. It consists of 7 ASCII bytes: 3C 48 55 31 30 2F 3E. According to the format above, we know that the 2-letter identifier of the part is HU. Looking up HU, in the mini tag table below, we find that "HU" is of tagged device type "Humidity". The 2-digit implementation number, NN, is "10". Although NN is subjective in nature, the number picked should have some attached meaning and in this case, stands for the last 2 digits of the humidity sensor's part number DS1910. For the mini tag <HU10/>, the CC and II identifiers are not needed and thus, not included.

Table 2. Mini Tag Type Table

Mini Tag 2-Letter Abbreviation	*Type*
CO	Contact (sensor)
EV	Event (sensor)
SW	Switch (actuator)
TH	Thermal (sensor)
HU	Humidity (sensor)
DA	D2A (actuator)

Mini Tag Translation

The mini tag obviously is not a complete XML document that can be parsed, but it can easily be translated into one. One way to do this is through mini tag translation. Mini tag translation takes the mini tag and translates it into a 1-Wire tagging XML document. For example, the mini tag, <HU10/>, can be translated into the following 1-Wire tagging XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="E200000006283826" type="Humidity">
</sensor>
```

The first step in translating the mini tag to an XML document is by pre-pending the following XML header to the beginning of the document: <?xml version="1.0" encoding="UTF-8"?>. The second step is to look up the mini tag's 2-letter type descriptor in the above table and retrieve the type attribute and parent element. In this case, the type attribute is "Humidity" and the parent element is <sensor>. The last step is to determine the *addr* attribute of the tagged device. This, of course, is the 1-Wire Network Address and is already known through the 1-Wire search protocol that occurred previously (by discovering the device and reading its memory contents).

Miscellaneous XML 1-Wire Tagging Examples

Two XML 1-Wire tagging examples can be found below. Example 1 shows a nested cluster of devices representing dimmable lights and adjustable radio volumes in different rooms of a building. Example 2 shows the nesting of branches. The first branch is a "Hub Switch", the second branch resides in a Weather Station cluster and implements a "Contact" sensor to determine if a North wind is blowing.

Example 1. Nested Clusters.

```
<?xml version="1.0" encoding="UTF-8"?>
<cluster name="Building C">
        <cluster name="Room C165">
                <actuator addr="BB0000000062602C" type="D2A">
                        <label>Dimming Fluorescent Light</label>
                        <channel>0</channel>
                        <scale>Light Intensity</scale>
                </actuator>
                <actuator addr="B2000000005AE32C" type="D2A">
                        <label>Radio Volume</label>
                        <channel>0</channel>
                        <scale>Sound</scale>
                </actuator>
        <cluster name="1-Wire Lab">
                <actuator addr="00000000005AE52C" type="D2A">
                        <label>Dimming Workbench Light</label>
                        <channel>0</channel>
                        <scale>Light Intensity</scale>
                </actuator>
        </cluster>
</cluster>
```

Example 2. Nested Branches.

```
<?xml version="1.0" encoding="UTF-8"?>
<branch addr="B700000018AE3212" >
        <label>Hub Switch #1</label>
        <channel>1</channel>
        <init>0</init>
        <cluster name="1-Wire Weather Station #2">
                <sensor addr="E200000006283826" type="Thermal">
                        <label>Outdoor Temperature</label>
                </sensor>
                <branch addr="77000000023CEC12" >
                        <label>Wind Vane Switch</label>
                        <channel>1</channel>
                        <init>0</init>
                        <sensor addr="0E00000200522401" type="Contact">
                                <label>North Wind</label>
                                <max>Making contact</max>
                                <min>No contact</min>
                         </sensor>
                </branch>
        </cluster>
</branch>
```

XML Parsing and Software Implementation

The above sections of this document provide data on what makes up XML 1-Wire tagging files and how to write and use them. With that in mind, we can now focus on the software implementation details. As of the time of this writing, the parsing software has been implemented successfully in Java, so the discussion will take place with the Java platform in mind.

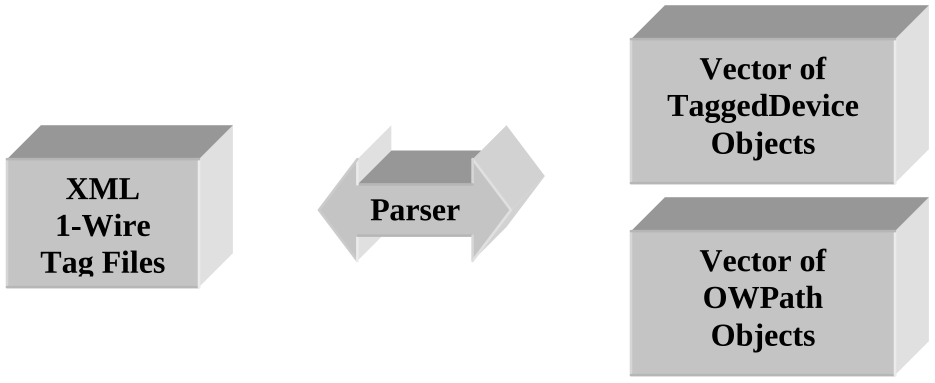
The Parser

As has been discussed above, the Java implementation of XML 1-Wire tagging is based on a SAX compliant parser. The reason for choosing this kind of parser is its lightweight implementation and thus its ability to run on very small handheld devices, such as TINI (http://www.ibutton.com/TINI). Any SAX parser can be used for XML 1-Wire tagging, but it is recommended that the parsers verified to work on the TINI platform be used. Two Java SAX parsers known to work on TINI are MinML (http://www.minml.com) and NanoXML (http://nanoxml.sourceforge.net).

Software Object Creation

As Figure 1 below shows, the SAX parser parses the XML 1-Wire Tag file, and generates two collections of software objects. The first collection is a dynamic array (vector) of software objects representing tagged devices. The Java class name for the tagged device software object is TaggedDevice. A TaggedDevice as explained earlier can be a branch, sensor or actuator. The second collection of objects consists of a vector of OWPaths. An OWPath can be thought of as a pathway to get to an actuator or sensor through a set of branches (in this case, 1-Wire switches).

Figure 1. Highest Level View of Object Creation



The TaggedDevice Object

The TaggedDevice comes in 3 flavors: branch, sensor, and actuator. A branch acts like a switch or a path, a sensor “reads” or “senses” a measurement or activity, and an actuator is a device that can be exercised or manipulated. All TaggedDevice objects have the same data members, but not all are used for each type of TaggedDevice. For a list of data members and a description, see the table below.

Table 2. TaggedDevice Data Members with Description

Data Members	*Description*
label	String equivalent to <label> child element.
max	String equivalent to <max> child element.
min	String equivalent to <min> child element.
channel	Integer equivalent to <channel> child element.
init	String equivalent to <init> child element.
scale	String equivalent to <scale> child element.
type	String equivalent to *type* attribute.
clusterName	A String representing the cluster path to the tagged device. If in a nested cluster, the clusterName is made up of all the clusterNames previous to it separated by “/”.
deviceContainer	Specific OneWireContainer for the tagged device.
branchPath	The OWPath to the tagged device. See OWPath discussion below.

The data members of TaggedDevice map very closely to the XML child elements discussed above. The first six data members represent the six child elements of which all but *channel* are strings. The data member, *channel*, should be an integer since it will be used as such.

The data member, *clusterName*, is a string that contains the name of the cluster to which the TaggedDevice object belongs. Since clusters can be nested, the clusterName is pre-pended with a path of clusters, if one exists, similar in nature to a path of a file on a computer. For example, “Building C/Room165/North Wall” is the valid clusterName for any TaggedDevice found in the “North Wall” cluster. This represents a nesting of clusters three deep. Notice they are separated by a forward slash, “/”.

The implementation of clusterName in Java is done through creating a stack of strings upon the startDocument event during XML parsing. Then, when a new <cluster> element is discovered during parsing, its text string is “pushed” on the stack, and when the <cluster> element ends, the <cluster> string is “popped” off the stack. All TaggedDevice objects that get created between the beggining and end of the <cluster> element take a snapshot of the stack and save it with “/” as separators into the clusterName data member.

The deviceContainer is a very important data member. It is a OneWireContainer that is specific to the 1-Wire device. This object is part of and is described fully in the 1-Wire API for Java (http://www.ibutton.com/software/1wire/1wire_api.html). This object gets instantiated for the TaggedDevice object when, during XML parsing, the *addr* attribute is encountered, which is equivalent to the tagged device’s 1-Wire Net Address. The OneWireContainer created for the device contains all the methods to completely access and exercise the tagged device.

Finally, the last data member is branchPath, the OWPath of the tagged device. As with the OneWireContainer object, the OWPath object is also fully described in the 1-Wire API for Java (http://www.ibutton.com/software/1wire/1wire_api.html).

The OWPath

An OWPath object is constructed with a OneWireContainer and a channel number as input parameters, both of which a TaggedDevice already has. Its purpose is to act like a path, similar in nature to a path of a file on a computer. Only the “path” here is the idea of a path through possibly many nested branches (1-Wire switches) to get to a particular tagged device. The branchPath of a specific TaggedDevice object, then, should be opened before a transaction with a TaggedDevice starts and closed afterwards. The act of opening the branchPath sets the 1-Wire switches to the appropriate state, enabling the tagged device to be exercised.

There are two places in this specification where OWPath objects are necessary. The first is in the TaggedDevice object, as mentioned above, and the second is in the vector returned from parsing an XML tag. In the Java implementation of this specification, the OWPath objects are created through keeping track of branches with a branch stack during XML tag parsing. When the TaggedDevice object is created, the stack of branches is cloned (basically, a “snapshot” is taken) and is made a data member of the TaggedDevice (as a vector of branches). This data member was not mentioned above since it is created for the sole purpose of eventually creating the branchPath of the TaggedDevice. Once the XML document is parsed, some post-processing takes place in the endDocument method of the SAX parser. Each TaggedDevice object is iterated through and its vector of branches is used to make the TaggedDevice object's branchPath.

As to the vector of OWPaths returned after parsing the XML tag, something similar takes place. For this, the same branch stack above is used. However, just before the branch stack is "popped" (from parsing the end of a <branch> element), the branch stack is cloned and is placed into a vector for safekeeping. Again, some post-processing occurs in the endDocument method of the SAX parser, and the vector of branch stacks is iterated through and made into OWPath objects. The OWPath objects are then stored in a separate vector that is eventually returned by the parser.

Extending the TaggedDevice Object

The TaggedDevice objects, returned by parsing an XML tag, should be extended TaggedDevice objects, with the exception of a branch. This means that they all contain the data members and methods associated with a TaggedDevice, but they may have additional data members and methods and/or method implementations unique to their type. For example, a "Contact" device will have different methods and/or method implementations than a "Switch" device. Please see the above section entitled "Types" for a list of types that have been implemented. Each type of device will have its own unique class associated with it. Thus, if more 1-Wire devices are designed that fall into a new type category, a new Java class will need to be written for it.

To abstract things further, this implementation specifies a sensor and actuator interface, TaggedSensor and TaggedActuator respectively. So, any new extended TaggedDevice object created must implement the appropriate interface. The TaggedSensor contains the readSensor() method which returns a string representing the most current reading of the sensor, and the TaggedActuator contains the following three methods, getSelections(), setSelection(String selection), and initActuator(). The first method, getSelections(), retrieves the state selections that can be changed or exercised on the actuator. They are returned as a vector of strings. The second method, setSelection(String selection), actually sets the state on the actuator given a string that matches one of the selections. And, finally, the third method, initActuator() initializes the actuator to a pre-defined state. This method should be called before calling the other two. For the Java implementation of this specification, each new type of TaggedDevice object will have the statement "extends TaggedDevice" in its source code and either "implements TaggedSensor" or "implements TaggedActuator" as well. For example, the source code of the "Contact" type of TaggedDevice has the following line, "public class Contact extends TaggedDevice implements TaggedSensor".

The Master Application

The master application is a program that exercises the tagging software libraries. It is responsible for interfacing with the user, for thread management, for 1-Wire synchronization, for catching 1-Wire exceptions, and for searching branches for tags and mini tags. An example master application is included with the Java implementation of this specification. It demonstrates good programming practices when using the software libraries.

The Tag Creator

Since XML is an open standard a tag-creation application is not essential to 1-Wire tagging, but it is strongly suggested that the developer create one to go along with his tagging application. A tag creator simply creates XML 1-Wire tags specifically for a developer's program. It should prompt the user for the appropriate tag elements, and when properly filled out, it should write the XML document or mini tag to the appropriate 1-Wire device or operating system file system.