Nov 16, 2001

APPLICATION NOTE 158

1-Wire Tagging with XML

This document will present a 1-Wire® Tag format in XML that describes the 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.

Introduction

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. 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.

Consider the scenario where two switches are found on a 1-Wire network. One switch is used as a branch beyond which other 1-Wire devices are located, and the other switch is used to open a high security door. To the 1-Wire master software, it is difficult to differentiate between the two devices without first exercising them. But, if the master software first exercises the two switches, it could inadvertently open the high security door. Without knowing the difference between the functionality of the two switches beforehand, the 1-Wire master software can, in the above scenario, compromise security. The recommended way to solve this problem is through 1-Wire Tagging.

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.

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 that can be handled by 1-Wire devices with small memory footprints, but the format is "home grown" and not known as an industry standard. 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.

This 1-Wire Tagging scheme requires that the reader know some of the basics of XML. For more information about XML, please visit the World Wide Web Consortium's (W3C) website, http://www.w3.org. W3C is the governing body for the industry standards that define XML. XML tutorials can also be found on the web at the following locations: http://www.w3schools.com/xml/default.asp and http://java.sun.com/xml/tutorial_intro.html.

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 a 1-Wire File Structure file. The file can reside in any normal memory 1-Wire device such as the DS1993, DS2406, or DS2433. The specification for the 1-Wire File Structure can be found in the Dallas Semiconductor Application Note 114, 1-Wire File Structure.

The 1-Wire 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 1-Wire 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 Tag Definition

A 1-Wire Tag is a valid XML data object (can be thought of as a file or data stream) that contains predefined 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™ 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 1-Wire Tag, make it as small as possible while maintaining human readability.
   a. Keep the 1-Wire Tag elements small in size.
   b. When writing tag files into 1-Wire devices, remove all unnecessary white space.
   c. Use empty XML elements, called mini tags, in small memory situations.
2. The master application should be able to parse multiple 1-Wire 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 1-Wire Tags, the master application should create a list of software 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. The master application should also return the 1-Wire Network's topology of branches and nested branches.
4. For communicating to the 1-Wire devices, use one of the standard APIs available. Three APIs are available: the 1-Wire API for Java, TMEX, and the 1-Wire Net Public Domain Kit. Information on each of these APIs can be found here: http://www.ibutton.com/software. The 1-Wire API for Java, (found at http://www.ibutton.com/software/1wire/1wire_api.html) will have a reference implementation of the 1-Wire Tagging scheme described in this document as part of its API in Version 1.00 Alpha and above. Please note that this reference implementation does not necessarily meet the specification found in this document in its entirety.
5. Parse the XML tags with an event-based parser, using the Simple API for XML (SAX) implementation. Since its footprint is small and does not keep large data objects in memory, using SAX will enable small devices (including TINI® and handhelds) to use the 1-Wire XML tagging format described in this document. Also, at the time of this writing, it has been verified that at least two SAX-based XML parsers run successfully on TINI. Thus, 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 XML elements that make up a 1-Wire 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 (DS2405, DS2406, DS2407, 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.

Example 1. 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 even other <branch> elements. Please note that 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.

Example 2. Sensor XML 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 Table 1 in 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.

Example 3. Actuator XML 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.

Example 4. Cluster XML 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 eight child elements are as follows: <label>, <max>, <min>, <channel>, <init>, <scale>, <hightrip>, and <lowtrip>. 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 are examples of this.

Example 5. Child Element Examples

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

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

The Contact Example 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 eight child elements as data members. The Switch Example shows an actuator of type Switch" and uses five of the eight child elements.

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. The Contact Example uses <max> and <min> to describe the two different possible states of the "Contact" tagged device. The Switch Example 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, the example below shows <max> being used by itself as an event message.

Example 6. Child Element <max> Usage

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

Table 1 below shows a list of types and the child elements used by each type. The "x" marks a "used" element.

Table 1. Child Elements Used by Type

Type	Child Elements							
	label	max	min	channel	init	scale	hightrip	lowtrip
Switch	x	x	x	x	x			
D2A	x			x		x		
Contact	x	x	x					
Event	x	x		x				
Level	x	x	x	x				
Thermal	x	x	x				x	x
Humidity	x	x	x				x	x
A2D	x	x	x	x			x	x
Counter	x	x		x			x	
Pressure	x	x	x				x	x
Date	x	x		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. Below is an example of the <label> child element.

Example 7. Child Element <label>

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

In the above example, the <label> element describes the "Humidity" 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 in Examples 4 and 5. Keep in mind that neither <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

Opposite to <max>, the <min> child element also 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 be represented as 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. In the example below, <channel> is "1" which will select the second switch.

Example 8. Child Element <channel> Example With Type "Event"

```
<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, the channel element is "0" (since, at the time of this writing no 1-Wire device contains multiple potentiometers).

Example 9. Child Element <channel> Example With Type "D2A"

```
<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 Example 1 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 10. Child Element <init>

```
<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 11 below, the tagged device of type "D2A" uses the child element <scale> to represent a volume scale in decibels.

Example 11. Child Element <init>

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

Hightrip
The <hightrip> child element is generally used for tagged devices, usually sensors, when a high trip value is needed for detecting high values. An example of its use would be to detect if a refrigerated air conditioning system needed to turn on when the temperature of a particular room got too hot. It can also be used for a clock alarm, for a counter alarm, and for humidity or pressure sensing.

Lowtrip
The <lowtrip> child element is generally used for tagged devices when a low trip value is needed for detecting low values of a condition being monitored or sensed. An example of its use would be to detect if a heating system needed to be turned on when the temperature of a particular room got too cold. It can also be used for humidity or pressure sensing. An example of <lowtrip> and <hightrip> can be found below.

Example 12. Child Elements <hightrip> and <lowtrip>

```
<sensor addr="E200000006283826" type="Thermal">
        <label>Indoor Temperature</label>
        <lowtrip>18.0</lowtrip>
        <hightrip>30.0</hightrip>
        <min>It's too cold, please turn up the heat</min>
        <max>It's too hot, please turn on the air conditioner</max>
</sensor>
```

User-Defined Child Elements
The programmer or end-user is not limited to use only the enumerated child elements above. One of the advantages to using XML is its ability to be extended. Using more elements should not negatively affect or impact this specification. In fact, using more child elements is encouraged. Some suggestions are: <manufacturer>, <date>, <netregistration>, and <enum>. The <manufacture> element could contain information on the manufacturer of the 1-Wire part or cluster/subsystem. The <date> tag, of course, would provide the date of manufacture. Many companies use the format of WWYY where W stands for the work week number and YY stands for the year. The element <netregistration> could contain a URL to the manufacturer's website, or possibly to a server where the software for the particular 1-Wire part or cluster can be automatically downloaded and run. Finally, <enum> could specify a serialization number that keeps track of the actual 1-Wire part or cluster, so each part or cluster that gets tagged receives a unique incremental number. This number could then be stored in a

database giving part traceability.

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, ten types have been identified: Contact, Event, Switch, Thermal, Humidity, D2A, A2D, Counter, Pressure, and Date. Please note that the value of type should always be capitalized in a 1-Wire Tag. 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.

The types of 1-Wire Tags can be classified into two groups: actuators and sensors (with branches being their own unique type). These groups directly correlate to the <actuator> and <sensor> parent XML elements. As mentioned previously, actuators are tagged devices that can be acted upon or can be exercised, and sensors are tagged devices that can monitor or sense a specific condition and give a reading when asked. Out of the ten types, two of them are classified as actuators, and eight of them are classified as sensors. The actuator types are Switch and D2A. Thus, if a 1-Wire Tag specifies the <actuator> element, its type will be either Switch or D2A. Conversely, if a 1-Wire Tag specifies the <sensor> element, its type will be one of the eight sensors: Contact, Event, Thermal, Humidity, A2D, Counter, Pressure, or Date. Please note that the examples listed in this section will be complete 1-Wire Tags with an appropriate XML header included for each example.

Actuator Types

The two actuator types identified by this document are Switch and D2A. The Switch type represents any device in the family of 1-Wire switches. Among them are the DS2405, DS2406, DS2407, and the DS2409. The D2A type represents any device in the 1-Wire family of digital potentiometers. Currently, there is only one part, the DS2890. Both the Switch and the D2A types, when parsed by the XML parser, will return software objects that, when queried, will give an array of state selections from which to choose. When a particular selection is chosen, then the state of the actuator is changed to the selection made. For the Switch type, the selections are either an open connection or a closed connection, and for the D2A type, the selections returned comprise an array of choices specific to the DS2890 representing digital potentiometer wiper positions.

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).

Example 13. "Switch" 1-Wire Tag

```
<?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>
```

D2A
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 software object of type D2A created as a result of parsing the XML file, is also a place where state selections can be determined. The software object, then, can be used to determine the wiper states from which to choose. The next child element, <channel>, represents the specific potentiometer on the 1-Wire Tagged device to be exercised. The <channel> child element should be represented as an integer. Finally, the <scale> child element is used to determine the measurement scale represented by the state selections. For the example below, the scale is "Light intensity".

Example 14. "D2A" 1-Wire Tag

```
<?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>
```

Sensor Types

The eight sensor types identified by this document are Contact, Event, Thermal, Humidity, A2D, Counter, Pressure, and Date. Contact makes use of any 1-Wire device, and its purpose is to sense if the tagged device is present on the 1-Wire bus. An Event sensor's purpose is to detect if activity has occurred on a particular 1-Wire switch. Thermal, as its name implies, is a sensor used to sense the temperature of its surroundings, and can be used with any 1-Wire digital thermometer. A partial list of them includes the DS1921, DS1920, DS2760, DS2438, DS18S20, and the DS18B20. Humidity senses humidity, and as of the writing of this document, pertains to the DS1910. A2D senses voltages coming from an analog-to-digital 1-Wire device. Among these are the DS2438, DS2760, and the DS2450. Counter sensors detect and count voltage pulses. Counter supports the DS2423, the DS2404, and the DS1994. Pressure, of course, is a sensor that detects pressure. It is listed in this document as a placeholder because, currently, no 1-Wire pressure sensor is available commercially. And, finally, Date is a sensor that detects elapsed time.

Contact

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. An example of a complete 1-Wire Tag of type "Contact" with the appropriate XML header is shown below.

Example 15. "Contact" 1-Wire 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

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.

Example 16. "Event" 1-Wire Tag

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

Level

A "Level" sensor has four child elements: <label>, <max>, <min> and <channel>. Again, <label> is used to describe the sensor. In Example 17 below, the cluster name attribute specifies that the indicated 1-Wire device is a DS2406. This DS2406 is actually a cluster of two tagged devices. The first is of type "Level", and the second is of type "Switch". For the "Level" sensor, the <label> element describes the sensor as a "Refrigerator Door Light". The <max> child element is the event message whenever the switch is in a conducting state. Thus, the associated message for the event would be "Light is on". Conversely, the <min> event message indicates that the switch is in the non-conducting state giving the message "Light is off". And, the <channel> child element represents the specific switch on the 1-Wire Tagged device to be sensed.

Example 17. "Level" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<cluster name="DS2406 Demo">
        <sensor addr="36000000000F2212" type="Level">
                <label>Regrigerator Door Light</label>
                <max>Light is on</max>
                <min>Light is off</min>
```

```
                <channel>0</channel>
        </sensor>
        <actuator addr="36000000000F2212" type="Switch">
                <label>Contact Maker</label>
                <min>Open Circuit</min>
                <max>Make Contact</max>
                <channel>1</channel>
                <init>0</init>
        </actuator>
</cluster>
```

Thermal
A "Thermal" sensor has three child elements: <label>, <lowtrip>, <hightrip>, <max>, and <min>. Of course, <label> is used to describe the sensor. In the example below, <label> indicates that this tagged device is sensing an indoor temperature.

Optionally, a tagged device of type "Thermal" can have trip points monitored both at a high level or a low level. Like a thermostat, it may be desirable to know when a temperature surpasses a given temperature and again when it drops below a given temperature. To do this, the <hightrip> and <lowtrip> elements are used to set the monitored high and low levels (as a floating point number), and the <min> and <max> elements are used to contain simple text messages to be used when the trip points are surpassed. See the example below.

Example 18. "Thermal" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="E200000006283826" type="Thermal">
        <label>Indoor Temperature</label>
        <lowtrip>18.0</lowtrip>
        <hightrip>30.0</hightrip>
        <min>It's too cold, please turn up the heat</min>
        <max>It's too hot, please turn on the air conditioner</max>
</sensor>
```

Humidity
A "Humidity" sensor's first child element is <label>. In the example below, <label> indicates that the specified tagged device is sensing an outdoor humidity.

Optionally, a tagged device of type "Humidity" can have trip points monitored both at a high level or a low level. It may be desirable to know when the monitored humidity surpasses a given value and again when it drops below a given value. To do this, the <hightrip> and <lowtrip> elements can be used to set the monitored high and low levels. These levels should be represented as floating point numbers. The <min> and <max> elements are used to contain simple text messages to be used when the trip points are surpassed. See the example below.

Example 19. "Humidity" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor addr="E200000006283826" type="Humidity">
        <label>Outdoor Humidity</label>
        <lowtrip>20.0</lowtrip>
        <hightrip>90.0</hightrip>
        <min>It's too dry</min>
        <max>Expect fog or rain</max>
</sensor>
```

A2D
An "A2D" sensor has six child elements: <label>, <channel>, <lowtrip>, <hightrip>, <min>, and <max>. The "A2D" sensor's first child element is <label>. In the example below, <label> indicates that the specified tagged device is sensing a voltage. Since A2D tagged devices have multiple channels on which to do analog-to-digital conversions, the element <channel> is used to determine which channel to read. Thus, the <channel> element is represented as an integer number. In the example below, the <channel> element indicates that the first channel, channel 0, is the one that should be read.

Optionally, a tagged device of type "A2D" can also have trip points monitored both at a high level or a low level. Consequently, when the voltage surpasses a given value and again when it drops below a given value, the <min> and <max> messages will be given. The <hightrip> and <lowtrip> elements represent the high and low values being monitored and should be given as floating point numbers.

Example 20. "A2D" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<actuator addr="E200000006283826" type="A2D">
        <label>Voltage Monitor</label>
        <channel>0</channel>
        <lowtrip>2.0</lowtrip>
        <hightrip>4.0</hightrip>
        <min>Voltage too low</min>
        <max>Voltage too high</max>
</actuator>
```

Counter
The "Counter" sensor is made up of four child elements: <label>, <channel>, <hightrip>, and <max>. The Counter sensor's first child element is <label>. In the example below, <label> indicates that the specified tagged device is acting as a rain gauge. Since some of the 1-Wire devices mentioned above have multiple counters, the <channel> element specifies which counter should be read. In the example below, the first counter, counter 0, is the channel specified. The <channel> element should represent an integer number.

Optionally, a tagged device of type "Counter" could also have a single trip point. If the counter goes above a certain count (the trip point), then an appropriate text message gets sent. In the example below, the <hightrip> element is set at 500 counts, so whenever the count surpasses 500, the message indicated by the <max> element gets sent that reads "Time to clean leaves out of rain gauge".

Example 21. "Counter" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<actuator addr="E200000006283826" type="Counter">
        <label>Rain Gauge</label>
        <channel>0</channel>
        <hightrip>500</hightrip>
        <max>Time to clean leaves out of rain gauge</max>
</actuator>
```

Pressure
The "Pressure" sensor is made up of four child elements: <label>, <lowtrip>, <hightrip>, <min>, and <max>. The Pressure sensor's first child element is <label>. In the example below, <label> indicates that the specified tagged device is sensing outdoor barometric pressure.

A tagged device of type "Pressure" can also have trip points monitored both at a high level or a low level. It may be advantageous to know when the monitored pressure surpasses a given value and again when it drops below a given value. The <hightrip> and <lowtrip> elements can be used to give the monitored high and low levels (as floating point numbers), and the <min> and <max> elements can be used as the source of simple text messages to be sent when the trip points are surpassed. See the example below. Currently, the Pressure type is just a placeholder, since at the time of this writing, no 1-Wire device directly supports pressure sensing.

Example 22. "Pressure" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
        <sensor addr="E200000006283826" type="Pressure">
        <label>Outdoor Barometric Pressure</label>
        <lowtrip>28.0</lowtrip>
        <hightrip>31.0</hightrip>
        <min>Low pressure</min>
        <max>High pressure</max>
</sensor>
```

Date
The "Date" tagged device is a sensor because its purpose is to read the time/date value and display it. In this case, the device could be one of many different 1-Wire devices. Among these are the DS1904, DS2415, DS1994, DS2404, DS1921, DS2417, and the DS2438.

Example 23. "Date" 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<actuator addr="020000002E5BD604" type="Date">
        <label>My iButton Clock</label>
        <hightrip>1000000</hightrip>
        <max>Time to buy another car</max>
```

```
</actuator>
```

The "Date" sensor's first child element is <label>. In the example above, <label> indicates that the specified tagged device is acting as an iButton clock. Optionally, a tagged device of type "Date" could also have a single trip point. If the time/date goes beyond a certain time (the trip point), then an appropriate text message gets sent. In the above example, the <hightrip> element is set at 1000000 seconds since 1970, so whenever the seconds surpasses 1000000, the message indicated by the <max> element gets sent "Time to buy another car". Please note that <hightrip> is a long integer representing the seconds since 1970, and the <max> element represents the text message when the clock surpasses the hightrip value.

Mini Tags

Due to memory constraints on 1-Wire devices, it is highly desirable to make 1-Wire XML tags as small as possible. In some extreme cases where normal XML tags are too large, this can be accomplished 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. It is recommended to use mini tags only if they are absolutely necessary. They part from XML industry standards in two ways. The first is that they are not valid XML documents in and of themselves (they are only valid XML elements). And, secondly, mini tags are not very human readable.

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.

For example, 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 Abbreviations

Mini Tag 2-Letter Abbreviation	Type
SW	Switch (actuator)
DA	D2A (actuator)
CO	Contact (sensor)
EV	Event (sensor)
LV	Level (sensor)
TH	Thermal (sensor)
HU	Humidity (sensor)
AD	A2D (sensor)
CT	Counter (sensor)
PR	Pressure (sensor)
DT	Date (sensor)

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. This process is called 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 found in Example 24 below.

Example 24. Mini Tag Translation of <HU10>

```
<?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).

XML 1-Wire Tag Examples

Two XML 1-Wire Tagging examples can be found below. The first example shows a nested cluster of devices representing dimmable lights and adjustable radio volumes in different rooms of a building. The second example 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 25. Nested Clusters 1-Wire Tag

```
<?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>
        <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 26. Nested Branches 1-Wire Tag

```
<?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>
```

Software Implementation

The above sections of this document provide data on what makes up XML 1-Wire Tags and how to write them. With that in mind, the details of an example software implementation that parses and uses the tags can be discussed. As of the time of this writing, a reference

implementation of the software has been successfully developed in Java with the 1-Wire API for Java, so the discussion will center around Java, and, thus, object-oriented terminology. The 1-Wire API for Java can be found and downloaded here: http://www.ibutton.com/software/1wire/1wire_api.html.

The Parser

As has been discussed above, the Java implementation of XML 1-Wire Tagging is based on a SAXcompliant XML 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, and two Java SAX parsers known to work well are MinML (http://www.minml.com) and NanoXML (http://nanoxml.sourceforge.net).

Software Object Creation

As Figure 1 below shows, the SAX parser takes the XML 1-Wire Tags, parses them, and, with the help of the 1-Wire Tagging software libraries, generates two collections of software objects. This is the highest level view of software object creation. The first collection is a dynamic array of software objects representing tagged devices. In the Java reference implementation, the class name for the tagged device software object is TaggedDevice, and a dynamic array is called a Vector.

The second collection consists of a dynamic array of software objects representing the network topology of the 1-Wire network, otherwise known as 1-Wire paths. In essence, they can be thought of as paths to get to specific actuators or sensors through a set of branches (1-Wire switches). In the Java reference implementation, the class name for this "path" object is OWPath.

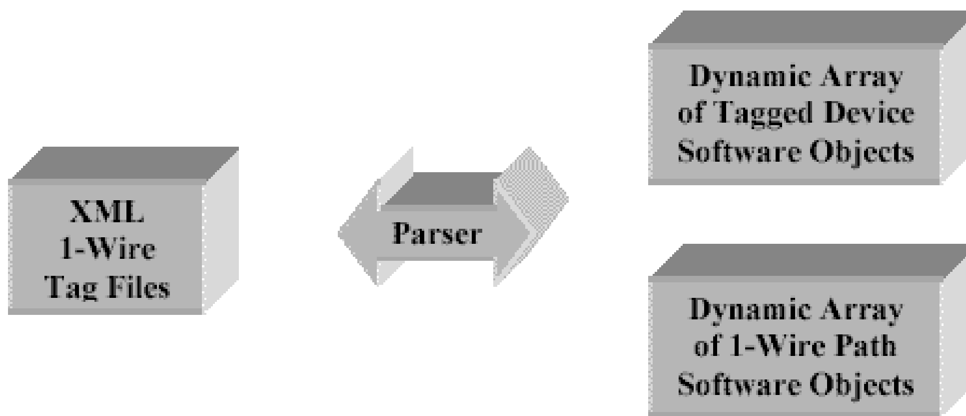

Figure 1. Highest Level View of Software Object Creation

The Tagged Device Software Object

The tagged device software object comes in three 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 tagged device software objects have the same data members, but not all are used for each type of tagged device. The Java implementation for the tagged device software object is "TaggedDevice". For a list and short description of the data members belonging to the TaggedDevice object, see the table below. All other software implementations should use at least the software equivalents of the data members listed below.

Table 3. 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.
hightrip	Floating point equivalent to <hightrip> child element.
lowtrip	Floating point equivalent to <lowtrip> 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 "/". For example, "BuildingC/Roof/WeatherStation#5".
deviceContainer	Specific OneWireContainer for the tagged device.
branchPath	The 1-Wire path to the tagged device. See the 1-Wire path discussion below.

The data members of the tagged device software object should map very closely to the XML child elements discussed above. This can be seen in the Java reference implementation of TaggedDevice shown in the above table. A discussion of the TaggedDevice follows.

The first eight data members of TaggedDevice represent the eight child elements of which many are strings except *channel, hightrip,* and *lowtrip*. The data member, *channel*, should be an integer and *hightrip* and *lowtrip* should be floating point numbers, since they will be used as such. The data member, clusterName, is a string that contains the name of the cluster to which the tagged device's software 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 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 of the TaggedDevice. It is a OneWireContainer that is specific to the 1-Wire device. It contains all the methods and fields necessary to completely access and exercise a specific tagged device. OneWireContainer objects are part of and are described fully in the 1-Wire API for Java (http://www.ibutton.com/software/1wire/1wire_api.html). The OneWireContainer 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.

Keep in mind that the OneWireContainer software object is only available in the 1-Wire API for Java. So, for other possible non-Java implementations of XML 1-Wire Tagging, the developer will need to use a collection of variables and functions in other 1-Wire software libraries to construct a software equivalent to deviceContainer.

Finally, the last data member of TaggedDevice is branchPath. This is the 1-Wire path of the tagged device. A 1-Wire path is a path to get to specific actuators or sensors through a set of 1-Wire switches. In the Java reference implementation, a 1-Wire path software object is called an OWPath. Similar to the OneWireContainer object, the OWPath object is also fully described in the 1-Wire API for Java. A discussion of 1-Wire paths and the Java equivalent OWPath follows.

The 1-Wire Path Software Object

The 1-Wire path software object's purpose is to act like a path, similar in nature to the 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 access a particular tagged device. The 1-Wire path software object should be made up of at least a few fundamental data members and methods. One important data member would be a collection of the 1-Wire net addresses of the various 1-Wire switches making up the paths. Another important data member would be a collection of the specific channel on each device to be exercised to access the desired sensor or actuator (since some 1-Wire switch devices have more than one switch per device). Also in the 1-Wire path software object, the methods for opening a path and closing a path should be included. The act of "opening" a path, means to loop through each 1-Wire branch device listed in the 1-Wire path and select the appropriate switch, and flip each switch to its "conducting" position. Conversely, "closing" the 1-Wire path would be flipping the appropriate switches to their "non-conducting" positions.

In the Java reference implementation for 1-Wire Tagging, the 1-Wire path software object is called the OWPath. This is what makes up the branchPath data member of the TaggedDevice object listed in Table 23 above. OWPath objects are also what make up the dynamic array (vector) of 1-Wire path software objects produced through XML parsing shown in Figure 1 above. The OWPath object is available through the 1-Wire API for Java, and the tagging reference implementation uses it extensively.

There are two cases in the reference implementation where OWPath objects are necessary. The first is in the TaggedDevice object, the branchPath, and the second is in the vector returned from parsing an XML tag. In the first case, each TaggedDevice's branchPath is created through keeping track of a branch stack during XML parsing. The stack mentioned here is the well-defined software structure called a "stack" with the usual accompanying methods of "push" and "pop". Thus, the branch stack is a stack of 1-Wire branches (or switches). When the TaggedDevice object is created during XML parsing, the current branch stack is copied to a temporary object belonging to TaggedDevice. Then, after the XML document is completely parsed, the resulting vector of TaggedDevice objects is iterated through and each branch stack stored is used to create the OWPath object.

In the second case where OWPath objects are necessary, something similar takes place. For creating the vector returned from parsing an XML tag, the same branch stack above is used. Only this time, the branch stack is copied to a temporary vector just before the branch stack gets "popped". After XML parsing completes, the vector of branch stacks is iterated through and used to create the vector of OWPath objects.

Extending the Tagged Device's Software Object

In object-oriented terms, the array of tagged device software objects returned by parsing 1-Wire Tags, should actually be an array of *extended* tagged device software objects, with the exception of a branch. This means that they all contain the data members and methods associated with a tagged device software object, 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 Table 1 above entitled "Child Elements Used By Type" for a list of types that have been identified by this document.

Each type of device will have its own unique tagged device software object associated with it. Thus, if more 1-Wire devices are designed that fall into a new type category, a new software object will need to be written for it. For the Java reference implementation, each "type" of tagged device has its own class file with the statement "extends TaggedDevice" in its source code. For example, the Contact type has a Contact.class written specifically for it. These software objects, then get invoked whenever the XML parses a 1-Wire Tag that specify them. As an added bonus, the Java reference implementation of 1-Wire Tagging invokes the software objects dynamically at run-time.

Sensor and Actuator Interfaces
To abstract things further, this document specifies that a sensor and actuator interface be used for each tagged device's software object. The term interface here is used in the Java sense of the term. An interface simply means that any software object that says it implements the interface must implement specific methods that the interface defines. For example, the sensor interface, called TaggedSensor in the Java implementation, consists of only one method, readSensor(). The readSensor() method defined in the interface returns a string representing the most current reading of the sensor. By implementing the interface, the software object is guaranteeing that it will provide a method called readSensor() that accepts no arguments and returns a string. Therefore, all tagged devices that are sensors, will be guaranteed to have the method readSensor(). For example, calling the readSensor() method in the sensor of type "Thermal", which in Java is in the Thermal.class, could return "23.5 degrees Celsius" as a possible return value.

Similarly, the actuator interface consists of a few specific method definitions. In the Java implementation, it is called TaggedActuator. The actuator interface consists of 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 dynamic array of strings, or a vector of strings in the Java implementation. 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 Java 1-Wire Tagging libraries.

Optionally, a 1-Wire Tag creation application could be created for a master application or its functionality could be used as part of the master application. A 1-Wire Tag creator simply creates XML 1-Wire Tag files specifically for a developer's program. It could prompt the user in wizard-like fashion for the appropriate tag elements, and when properly filled out, it could write the XML document or mini tag to the appropriate 1-Wire device or file. There is an example one available in the Java reference implementation of 1-Wire Tagging.

Conclusion

This document, when implemented in software, not only solves the general problem brought up by the introduction section of this document, but also enhances and simplifies the use of 1-Wire devices in real world applications. The problem in the introduction described a scenario where two switches are found on a 1-Wire network. One switch is used as a branch beyond which other 1-Wire devices are located, and the other switch is used to open a high security door. In this situation, the problem is the difficulty in differentiating between the two devices without first exercising them and inadvertently opening the high-security door in the process.

XML 1-Wire Tagging is the solution. Here is an example tag that would differentiate between the functionality of the two switches beforehand.

Example 27. High Security Door 1-Wire Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<cluster name="Area 51">
        <branch addr="2700000016EF3A12" >
        <label>Secure Room #1</label>
        <channel>1</channel>
        <init>0</init>
        <sensor addr="490000000212D016" type="Contact">
                <label>Government Badge #52</label>
```

```
                <max>Making contact</max>
                <min>No contact</min>
        </sensor>
                .
                .       (Other possible branches, actuators, or sensors)
                .
        </branch>
        <actuator addr="B700000018AE3212" type="Switch">
                <label>High Security Door: Alien Room</label>
                <max>Security cleared, door opened!</max>
                <min>Door shut and locked!</min>
                <channel>0</channel>
                <init>0</init>
        </actuator>
</cluster>
```

As can be seen in Table 30 above, the high-security door problem disappears. The first switch is shown to be a branch and clearly labeled as "Secure Room #1". Beyond the switch, we know that a Contact sensor is provided which, again is clearly labeled as "Government Badge #52". Any number of 1-Wire sensors, actuators, or branches could be grouped on the other side of the first switch.

The second switch is clearly marked as a "High Security Door: Alien Room", and it is identified as an actuator of the type "Switch". According to the 1-Wire device's XML elements, we know that the door is behind channel 0 (the first switch on the 1-Wire device) and that the switch on channel 0 is initialized to 0, or "non-conducting" when the program starts up. This means that the door is shut and locked upon program startup.

In the scenario above, not only does the problem get solved, but other desirable features such as grouping and even database-like functions get shown. For example, the entire 1-Wire Tag is grouped together as a cluster, entitled "Area 51". This shows that the 1-Wire Tag is easily extended to other 1-Wire clusters or groups. It also provides for 1-Wire network topology definition including the "nesting" of switches, and it provides simple database-like features, for instance, like keeping track of a list of people who have access to a high-security door. In conclusion, not only does 1-Wire Tagging solve a problem, but it enhances the developers and end users use and deployment of 1-Wire devices with the help of an industry standard, XML.

1-Wire and iButton are registered trademarks of Dallas Semiconductor.

More Information

DS18B20:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS18S20:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS1920:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2404:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2405:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2406:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2407:	QuickView	-- Full (PDF) Data Sheet	
DS2409:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2423:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2438:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2450:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples

DS2760:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS2890:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples