Alarm and Heating Controller

Application Note:

Using The AAG TAI 8558 One-Wire Multiple I/O Board:

Four bits of output via single pole, double throw, computer controlled relays, and
Four bits of opto-isolator protected inputs, which can be set with AC or DC voltages.

Version: 10 July 04

In this Note: All of the hardware issues surrounding using a TAI 8558 interface unit to....

a) provide security for a home, and

b) to use information from the security system to reduce the heating costs for the the home.

Along the way, there are may asides about more general 1-Wire points. The system is not intended to "the" answer to everyone's needs. It was devised to illustrate points which may excite the reader's imagination, and enable him or her to create something tailored to their particular needs. For example, while the system described below will lower the heating in a house when the owners are out, it would be trivial to revise it to reduce the money spent air conditioning a house in a hotter climate.

The Note does address software issues, but in a less complete manner, as either you know most of what you need to know, or no note of reasonable length will be able to give you what you need!

In addition to this note, you can obtain the following from the web:

Compiled program, will run under Windows 95 and higher.
Source code (Delphi)
Help file for the compiled program

The first is in ssds032se.exe
The second is in ssds032sc.exe

Both files are self-extracting zip files. Neither will do anything except unpack files into the folder of your choice. Nothing is done to your registry, menus, etc.

The third is included in the zip file of both the first and the second.

The file you are reading is available from AAG, or from the site below as ssds032.pdf

They should be found at http://sheepdogsoftware.co.uk, i.e. for the sourcecode enter....

```
http://sheepdogsoftware.co.uk/ssds032sc.exe
```

If you have trouble finding them there, check www.arunet.co.uk/tkboyd/offers1n.htm, or use Google to look for...

"sheepdog software" boyd

(Sheepdog Software is a registered trademark of TK Boyd)

Author of this note: TK Boyd.

1-Wire information by him at: http://www.arunet.co.uk/tkboyd/e1didx.htm

And, also by him, tutorials on the Delphi programming language for Windows, including 1-Wire issues at:
http://ourworld.compuserve.com/homepages/TK_Boyd/Tut.htm

The author is not an employee of AAG, the designers, manufacturers, trademark and copyright holders for the TAI 8558. AAG may be found at http://www.aag.com.mx. Nor is he employed by Dallas, the source of the 1-Wire product line.

This note, the graphics in it, and the .pdf version were all created using only the free, multi-platform suite called Open Office from www.openoffice.org.

Table of contents:

1. Miscellaneous terminology, naming of parts

2. Hardware

3. Software
    Low level subroutines
    Building the low level subroutines into the useful system

Appendix A: An explanation of some terms
Appendix B: Powering the TAI 8558
Appendix C: Other uses for inputs and outputs
Appendix D: Basic 1-Wire communications

Some terminology, etc:

Before I start, let me establish how I am going to describe the components on the TAI 8558. If you have one in front of you, turn it so that the LEDs are on the side that is facing you. Rotate it until the four large discrete components (2 caps, voltage regulator and a bridge rectifier) are on your left as you look at the board. Turned thus, you should see one "odd" LED in the "lower left" of the board. Across the "bottom" edge of the board are four screw terminals, with three connection points each. I will from time to time use the component ids in the silkscreen of the unit in front of me, but remember that these are subject to change, so take them with a pinch of salt.

I should also make clear that, even though you may have obtained this from the AAG site, nothing in this document can be taken as "definitive". For example, I say that the relays are rated 125VA. They may well be... but the relay specification is not set in stone, any more than anything else is. If in doubt: Check! The relays in the unit in front of me are clearly marked NEC EC2-5NJ, so I could easily go to NEC's website and check their specs.

Please do not attempt to work with household electricity unless you know what you are doing. This application note assumes that you will be working with low voltages. Any fires or deaths you cause by failing to take proper precautions with higher voltages, will, I'm afraid, have to be on your conscience and lawyer's books, not mine.

"MicroLan" and "1-Wire" are registered trademarks of Dallas Semiconductor, a division of Maxim. Website: www.ibutton.com

==================

Hardware:

Introduction to TAI 8558:

To build our home security / control system, what we need to know about the TAI 8558 is...

- It needs to be connected to a computer.
- It needs to be connected to a source of 12v DC. That power supply will also be tapped to sound the burglar alarm's sounders, so be sure it has sufficient power.
- It provides 4 relays (See entry in Appendix A if you are not familiar with these devices.) They can be switched independently, have NO and NC contacts, and are rated for 60W or 125VA.
- It provides 4 inputs, each protected by an opto-isolator. These are not high impedance inputs, as in TTL or CMOS work. You will create a small current, AC or DC through (either way, in the case of DC) the terminals for any of the inputs, and the computer, via the MicroLan, will be able to "see"

you have done so. (There's more on the connections in Appendix A, under the entry for "opto-isolator".)

I'll say more about the software side later, but for now I will just explain that the controlling computer can send messages to the TAI 8558 to turn any of the relays on or off, and it can read the state of the four inputs.

Required other parts:

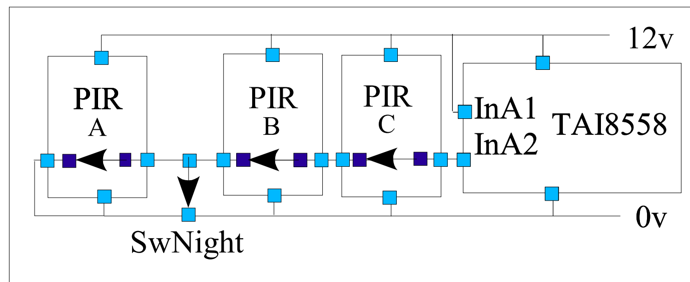
In order to create the system I proposed at the start of this, a few other things are needed:

At least one electronic sensor which can detect the presence of a "bad guy" in your house. This will typically be a PIR unit. PIR Units usually require a 12v supply, but that will be no problem for our project as the TAI 8558 needs 12 volts itself. Just be sure to choose a supply which can cope with all of the demands we are going to place on it.

Because PIRs usually have "closed = no intruder" "switches" in them, having multiple zones supervised is no problem: You just wire the PIRs in series.

Remember that the inputs on the TAI 8558 need current to flow to register an input.

Given all of the above, I hope the following makes sense. Ignore "SwNight" for the moment:



The circuit on the left shows a time when none of the detectors "see" an intruder, and, because of SwNight, even if PIR A does see one, the system won't know about it. I will explain "InA1", "InA2" later.

The drawing shows a TAI 8558 and three PIR units. All 4 are taking power from a 12 supply. The TAI 8558 is wired to that supply either through the RJ-12s or J1. (See Appendix B below, "Powering the TAI 8558"). Also from the 12v supply, we need a wire to one of connectors for one of the TAI 8558's inputs. From the other connector for that input, a wire goes to one side of the switch inside the first PIR ("PIR C" in the illustration.) From the other side of the switch in the first PIR, a wire goes to the switch in the next PIR, etc, etc, until you have all of the PIRs wired together in series. From the second connection of the last PIR's switch, you connect a wire to the 0v side of the power supply. Remember: The switches inside the PIRs will be closed if the PIR does not detect an intruder. Thus, as long as there is no intruder... and no broken wire... an LED in the TAI 8558's opto-isolator will be on. If any PIR detects an intruder, it will open its switch, the circuit will be incomplete, the opto-isolator's LED will go off. The supervising computer can "see" whether the LEDs inside the opto-isolators are on or off.

So what's "SwNight"? It may be an "ordinary", mechanical toggle switch. It may be the switch in one of the TAI 8558's relays. In any case, it's role is as follows. Imagine a fairly ordinary two story family

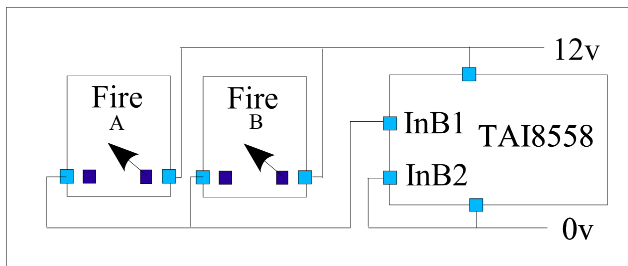
house. Do you only want the burglar alarm to protect the house when no one is at home, or would it be nice to have parts of the system "watching" when the family is asleep? Suppose PIRs A & B are in downstairs areas, but C is on a upstairs landing. By closing SwNight before arming the system, you can make it ignore any "person seen" messages from PIR C during the night. When the house is going to be empty, open SwNight, and PIR C once again becomes a way to trip the alarm.

(Throughout this document, I use "trip the alarm" to mean set the loud "something's wrong" bell wailing. I use "set the alarm" to mean make the system enter the state in which it will trip the alarm if it "sees" something wrong, and "unset the alarm" to mean return it to the state in which people can move about the house without tripping the alarm.)

There are other scenarios which are dealt with in the same way. I know someone who sometimes leaves her home for periods of a week or so. Neighbors water plants in her greenhouse when she's away. Her "PIR C" is the one that covers the greenhouse.

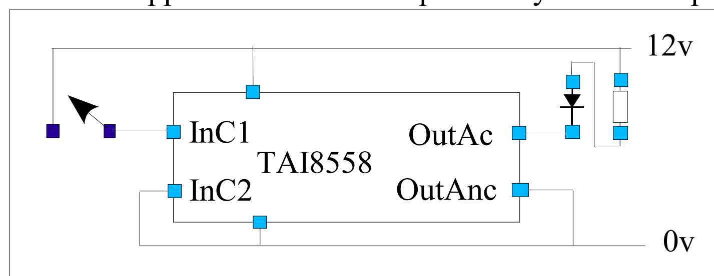
PIRs usually have anti-tamper switches. Their utilization is left as an exercise for the student!

While you are installing an alarm anyway, you might want to consider fire sensors. Smoke detectors are much more important to saving lives than heat detectors, but "mere" heat detectors have their place too. Some would say that a burglar alarm should not share a bell or siren with a fire alarm. In an imperfect world, maybe a shared bell is better than no bell. You can always write the software to make the bell ring continuously for fire, and in a pulsed mode for intruders. If you can remember the difference between two bells, you can remember the difference between one bell ringing two ways. And one final observation on bells: A very loud bell *inside* the protected premises can hasten the departure of intruders. It won't always work, but if it works sometimes, maybe it's worthwhile?



So. Back to the electronics. In the diagram, neither sensor is "seeing" a fire. You will probably find that your heat detectors do not conduct electricity until they "see" fire, which causes them to start conducting. This is the opposite of the behavior of a typical intruder detector. Fire detectors, through supervised circuits (see Appendix A) are protected from failures arising from a wire being broken. Simple "conducting when all's well" intruder circuits are vulnerable to bypasses like SwNight (see above) making parts of the detector array "blind", but they avoid problems caused by broken or cut wires without the extra complexity outlined in the appendix entry.

Simple input and output. I have no wish to insult anyone's intelligence with this diagram, but I also want this application note to help as many readers as possible, even beginners. On the left of the TAI 8558, you can see how to attach a simple switch to inputs on a TAI 8558. In the system we are discussing, you might want such a switch for setting the alarm. It is not always convenient to wake the computer's monitor just to see what you're doing with the keyboard. Also, using a switch makes it possible to have the arming take place where it suits you, not where the computer fits. Of course the switch must not be obvious to the wrong people if arming and disarming is to be achieved merely by throwing a toggle switch! More complicated protocols for changing the alarm's state can be created in the software. Alternatively, you can buy switches which are operated by a key, or modules with a keypad. The module has a relay in it, and the switch in that (to be wired into the TAI 8558 as the switch on the left above) is closed (or opened) when you enter the right sequence of numbers. Yet another option: The computer can be in a fairly secure place, but have a numeric keypad attached via USB. Such keypads can be purchased for about $20. As long as nothing was typed on the keypad or the main keyboard to deprive the supervising program of focus, the PIN for turning the system on or off could be entered without needing the computer right beside the door! Using PINs embedded in the software opens up the option of having different PINs for different authorized visitors, and logging who set / disarmed the alarm when.



If you want to be able to arm the system from two places, say near the exterior door for when you are going out, and on the upstairs landing for putting the alarm on in "night" mode, you simply add another switch.... but whether you put it in series or in parallel depends on what you want to set the alarm. With software being part of the system, you can easily define "set it" either way. I would suggest that "set it" be programmed to arise when 12v is NOT reaching InC1. To add a second switch for this would mean putting it in series with the first. Now if either switch is open.... or if a villain has cut a wire... the alarm will be watching for an intruder.

Cast your mind back to "SwNight" discussed earlier. If the switch on the upstairs landing is a double pole, double throw switch, then you can wire both the "turn alarm on for the night" and "bypass the upstairs PIRs" into, mechanically, "one" switch. Just be sure to incorporate some "exit time" into the programming, in case the "set alarm" message arrives slightly before the bypass circuit is complete. (Exit time: discussed more fully in software section.)

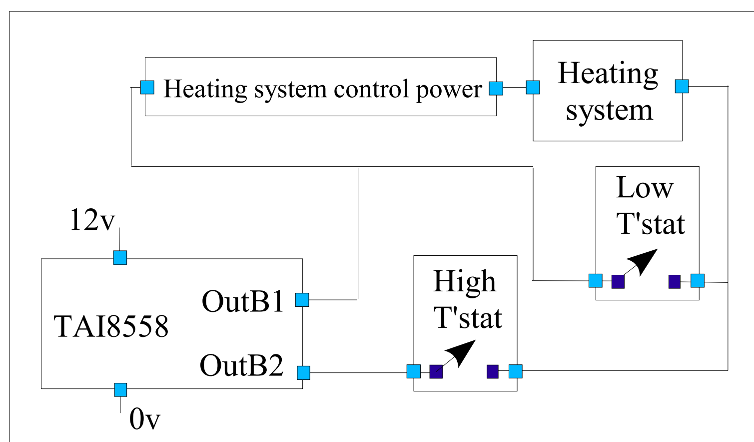
On the right in the figure above, I have tried to show a resistor and an LED. The TAI 8558 already provides you with LEDs, of course, but if you want something you can see in a different location, this is how you would wire one into the system. Because the TAI 8558 gives you relay protected outputs, there's nothing complicated about selecting your LED and resistor.... Inside the TAI 8558, effectively, there is just a simple switch between OutAc and OutAnc. In our alarm / heating control system, I would welcome an LED saying "alarm is set" which I could see through a window as I left.

Obviously, the system will require at least one loud bell or siren. That would be attached just like the LED and resistor, above. I would also suggest attaching on one of the other outputs a small piezo

sounder or buzzer to "communicate" with the user. For example, I'd like to hear some quiet beeps after I have flipped the switch to set the alarm system. One long beep when the alarm is disarmed would also be welcome. In some circumstances, you might want the quiet beeper to say "The main alarm bells are going to be ringing in 20 seconds if you don't do something about it before then." An easily remembered signal for that condition is the quiet beeper sounding "SOS" in Morse code. Of course, if you are wanting outputs for other things, the same sorts of messages can be conveyed via the "system armed" LED discussed earlier. Some easily distinguished patterns of beeps or flashes: A rapid sequence of short flashes and long dark periods, a slow sequence of long flashes and short dark periods. If you ring the changes of those ideas, you get four possibilities in addition to the "SOS", which should suffice. If you feel the need for more, you can work with 2, 3, 4 etc short flashes followed by a long dark period, with that repeating for as long as you want to display the message. Before you get too excited about all of this, you should know that for a simple system, you are putting a strain on the software (while it's busy sending a message by flashes, it is difficult to keep it tending to other things as well, which it must do under Windows, at least). Also, you should remember that there is a bit of latency in the 1-Wire system. As we are using it, it cannot switch things on and off at high frequencies. (On-off-on-off at one second intervals should be possible, but don't try to achieve audio frequencies, for instance.)

Hang in there! We've nearly finished the first part of this. You can skip down to "Connecting to the TAI 8558" if you're not interested in having the system affect your home heating (or cooling).

I love my computer for email. I love it for lots of things. However, I live in a region where if my heat were "stuck" "on" ***or*** "off", the consequences would be pretty awful. I don't think I have enough faith in my computer to entirely trust it with keeping my pipes unfrozen. Thus the "complicated" circuit below. Before starting on that, let me add that I did not incorporate a 1-Wire temperature sensing chip in this system, which you might well want to do, simply because the system is somewhat hypothetical, created to illustrate ways to use a TAI 8558.



First, just be be sure we're clear, I'll expand a bit on the labels in the circuit to the left.

The box marked "heating system" is standing in for the furnace, the pumps or fans, etc, etc. When electricity flows through it, the house gets heated.

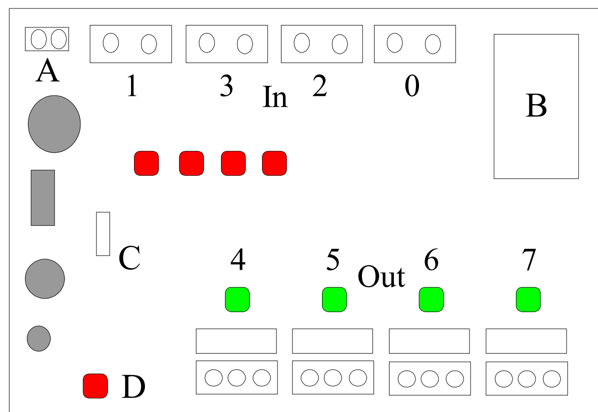
The box marked "Heating system control power" stands for a supply of a low voltage which turns the heating system on or off. Before our system is added to the house, there would be just these two elements, plus a thermostat wired as "Low T'stat" in the above. Note that I said *low* voltage. If your system is controlled by a high voltage, use something like the above, but in place of "heating system" add a relay "between" your circuits and the actual heating system. If you have to take this route, then the relay can be driven by the same 12v that is driving the TAI 8558. If you take this route, though, you'll need to give careful

thought to making it fail safe. The "Low T'stat" will probably need to be moved to the high voltage side of the circuit. Note also that if your system *does* use low voltage control that there is no connection between the TAI 8558's power and the heating system's. (The heating system can be AC or DC controlled... it doesn't matter, because between OutB1 and OutB2 there is merely the switch inside a relay.)

The diagram shows two thermostats. These are ordinary thermostats, in fact "Low T'stat" will probably be your system's original thermostat. They would be mounted beside one another. "Low T'stat" would be set to turn "on" at the temperature you want the house kept when everyone is in bed, or out. "High T'stat" would be set for the temperature you want when you are at home.

What is the point of all this complexity? Consider what happens if the software driving the TAI 8558 fails. It could leave the connection across OutB1/B2 either open or closed. The "worst" things that can happen are that the house will be as warm as it would be if you were home, even though you aren't, or that it will be a little cool, even though you are at home. A less complex system could result either in frozen pipes or the furnace running 24 hours a day if the system failed when you weren't there to correct the failure. Worth the complexity, I think!

Connecting to the TAI 8558:



It is time for a little tour of the TAI 8558. We've talked about what it offers. This discussion goes into the question of where specific items are to be found, and into the good range of options created for you by the board's designers.

If you have one, get it out, compare it to the diagram on the left. You should turn the board so that the integrated circuits are on the bottom, away from you.

I am not using the AAG component ids because they might change, even though I have no reason to believe they will.... I'm just being careful.

In the diagram there are three gray circles and one gray rectangle. They stand for components we don't need to discuss.

- - - -

The four two-connection screw terminals across the top of the board are the device's four inputs. Their numbers in the diagram show which bit they set or clear in the byte we will eventually use 1-Wire to read from the device. (More on reading the bits later.) The row of four red LEDs near the center of the device tells us the current state of each of the inputs. The one one the left reports on input "1", the next one reports on input "3", and so on. The LED will be on if there is current through the board, which gives rise to the DS2408 reporting that input as *LOW*. This may counter-intuitive, but it would complicate the TAI 8558's circuits to achieve the opposite behavior. You want it inexpensive and

reliable, don't you? It is very easy to cope with the situation in the software accessing the device.

In other parts of this document, I refer to input connections as InA1, InA2, InB1, InB2, etc. The A, B, C, D refers to the four available channels, which are marked 1, 3 , 2, 0 on the diagram above. It doesn't matter which channel you use for A or B or C or D. The 1 or 2 in "InA1", etc is simply to distinguish between the two connections available to each channel. Because the opto-isolators have two LEDs, one facing each way, it never matters which way 'round you connect things to the different channels. Note, however, that unless you add connections externally, there is no connection between either of the pins in, say, channel "A" and either of the pins of, say, "B". This is good! You can connect them if you need to, or have the information from independent circuits shared by the overall system through the software without any electrical interconnection.

For the intermediate reader, let me point something out: These "inputs" are not inputs like the high impedance inputs used, for instance, in TTL work. Your circuit must provide for current to flow through the LEDs inside the opto-isolators on the board, which are connected, for instance, to InA1 and InA2. The TAI 8558 does provide a 220 ohm current limiting resistor in series with the opto-isolator, so you can directly connect up to QCALC IT volts DC, or... QCALC volts AC. If you want to connect higher voltages, just be sure to add to the resistance limiting the current through the LEDs, and remember what I said about working with dangerously high power systems. Note I said high *power.* If your 5 volt power supply can source 100 amps, it is dangerous, even though voltage is low.

N.B.: The four red LEDs only operate if the jumper marked "C" in the diagram is in place. If you take away the jumper, the LEDs do not light. You may want to remove that jumper if you are using the device simply to read up to four inputs. As long as you do not want any outputs, and you can do without the LEDs reporting the state of the inputs, you do not need to supply the device with external power which is usually needed. (I.e. the *other* power in addition to the limited power that can be taken from the 1-Wire Data/Power line by the system Dallas calls parasitic power. It does work... but you have to be careful not to over-tax the overall system.)

\- - - -

The four three-connection screw terminals across the bottom of the board are the device's four outputs. Again, their numbers in the diagram show which bit in the byte we will eventually use 1-Wire to write to the device controls them. More on this later. The green LED just above each of the relays above the three connection screw terminals tells you whether that output bit is set high or low.

The TAI 8558 has a simple power on reset to ensure that the outputs from the DS2408 are all low shortly after power is applied. When an output from the DS2408 is low, the corresponding relay coil is *not* energized, and the LED is *not lit.* It may be that the outputs can be high briefly before the reset has been processed, but I doubt it would last long enough for the relay to be pulled in. Note also that if the 1-Wire signal from the master computer has not been lost during the time that all other power to the board has been lost, then the DS2408 does not go through a power-on sequence (it has drawn power "parasitically" (Dallas's term) from the MicroLan while the other power was absent.

In other parts of this document, I refer to output connections as OutAc, OutAno, OutAnc, OutBc, OutBno, OutBnc, etc. The A, B, C, D refers to the four available channels, which are marked 4, 5, 6, 7 on the diagram above. It doesn't matter which channel you use for A or B or C or D. The "c", "no" or "nc" in "OutAc", etc is simply to distinguish between the three connections available to each channel.

The middle connector on each of the connectors for the relays connects to the "common" pole of the relay's switch. The connector on the left (i.e. nearest to the "power on" LED) goes to the "NC" (normally closed) pole, and the connector on the right goes to the "NO" (normally open) pole.

Note: Unless you add connections externally, there is no connection between any of the pins in, say, channel "A" and any of the pins of, say, "B". This is good! You can connect them if you need to, leaving all of the driven circuits isolated from one another otherwise.

\- - - -

LED "D" in the lower left glows if the device's on-board voltage regulator is generating 5 v. This will happen if a suitable voltage (12v recommended... AC or DC will do) is supplied to the board via the two connection screw terminal marked "A" on the diagram (upper left). If the 12v is DC, it does not have to be regulated, and it doesn't matter which way 'round the connection is made, i.e. the positive lead can go to either post of the screw terminals marked "A". (*Not* "InAx" or "OutAx". The other A!)

You do not have to supply voltage via screw terminal "A". Your other alternative (apart from the rarte circumstances when you can do without it, as mentioned above) is to supply a regulated 5v via the connections at "B" This "solution" is a little risky: be sure you know what you are doing. *Whatever you do, don't connect something other than a* ***regulated*** *5 v this way.*

\- - - -

The connections at "B":

There are two possible configurations for the device. You may have one with a three connector screw terminal at "B". Alternatively, you may find one or two RJ-12 sockets at "B". If there are two, they are interchangeable. You plug the wire to the master computer (directly, or daisy chained by way of other 1-Wire devices) into one of the sockets. The other is for continuing the chain of devices onward away from the master computer.

If you have the RJ-12s:
The two central connections on the RJ-12 are for the data ground, and for the 1-Wire Data/Power signal, aka 1-Wire (or "OW") IO. I.e., these two are for the wires which make for the "1-Wire" network.

Lets agree on how we are numbering the contacts ("pins") in the RJ-12 socket: Hold the board up so that you are looking *into* the socket, with the contacts in the top of the cavity. The contact on your left is what I am calling "pin 1". The next one is "2", the next "3", etc.

The 1-Wire Data/Power signal is on pin 3. The 1-Wire data ground, or return, is on pin 4. I believe these two pins are used this way in every 1-Wire device I've seen. It is when you get to the other pins that things get interesting.

BEWARE: Not all of the people who build 1-Wire devices agree on the best use of the pins in the RJ connectors! You can't just plug things into each other without checking to see what is on each pin. That's the bad news. The good news is that, ***as far as I am aware*** (please alert me to any exceptions you've met!), everyone has always used the center two pins for the two fundamental "1-Wire" needs, and used the the same way 'round. This is true whether the connector has 4, 6 or 8 pins. Thus, you can

often plug a 4 pin plug into a 6 or 8 pin socket... especially if the plug only has connections to the center two contacts.

In the TAI 8558, the 1-Wire data return signal is on pin 4, as stated above. It is *also* on pin 2. In other words, those two pins are shorted together on the pcb.

Pin 1 *can* be used to supply the device with a regulated 5v DC. If this is done, then no voltage need be supplied via the two connection screw terminal identified as "A" on the diagram. If voltage for the on-board regulator is being supplied via "A", then there's no need to supply 5v via pin 1.

Pins 5 and 6 are not used by the device, but any signal on either carried forward between the two RJ-12's. I.e. pin 5 of the first RJ-12 is connected to pin 5 of the other RJ-12, and the same situation obtains for the pin 6s.

In the case of a TAI 8558 fitted with a three connection screw terminal in location "B", the connectors can be numbered as follows: Call the one near the top of the board, i.e. the one *least* close to diode "7", "1". Call the middle one "2", and call the lower one "3".

By that numbering, you should make connections as follows:

1: Used for the same purpose as pin 1 of the RJ-12. (Power feed. Regulated 5v only. See above.)
2: 1-Wire Data/ Power (OWIO)
3: 1-Wire data ground (OWRtn)

____

That pretty well covers the hardware aspects of the system. There are some further ideas for ways to use inputs and outputs in Appendix C, if you decide you don't need some of those proposed in the basic system, or if you decide to add a second TAI 8558 to your system. The ease with which you can add further 1-Wire units to a system is, of course, one of its greatest virtues.

============

Software:

I am going to start by discussing the low level routines I will make for the device before going on to explain the software for the system this Application Note is presenting. This makes for some pretty heavy reading before you get to the fun stuff at "The fun begins". If you're only reading this out of curiosity, skip down to there! The program is called DS032.

In any TAI 8558 program I write, you will probably find the following called once as the program begins to execute:

```
procedure DMLPreInit;
procedure DMLInit;
```

The prefix "DML" is for "Dallas MicroLan".

DMLPreInit fills three global variables: One holds the number of the port the 1-Wire adapter is on, one holds a code for the adapter type, and the last holds the id of the DS2408 in the TAI 8558. Yes, I know that global variables are a Bad Idea. I think there are times to break rules, this being one of them. Parts of the program that comes with this Application Note are written with a view to one day expanding the program to allow access to multiple one wire chips. When that day comes, DMLPreInit will also have to supply a value for how many chips are in the system. At that time, the ids of the chips will go into an array.

DMLInit needs the information prepared by DMLPreInit, and goes on to initialize further global variables needed by the routines which access the DS2408.

To do anything over the 1-Wire, you first "Establish a Session". If you are curious about the details of this, see Appendix D.

Two further basic routines make up the lowest level of the program. Each will establish a session (and release it on exit). One reads what is present on the DS2408's inputs, the other makes the outputs send whatever is needed. The declarations of these procedures are as follows:

```
procedure DMLGetPIOLogicState(bIndex:byte;
        var bAns:byte;
        var sTKBErr:string;
        var iDalErr:integer);

procedure DMLWriteToEnabledOutputs
        (bIndex,bValue,bMask:byte;
        var sTKBErr:string;
        var iDalErr:integer);
```

For those readers who do not read Delphi / Pascal, I'll go through the first one in detail.

Suppose your program had the following fragment of code....

```
procedure DMLGetPIOLogicState(0,bBitStates,sErr,iDErr);
```

Between the way Delphi works and what is in the procedure DMLGetPIOLogicState, after this line was executed, you would find new values in the variables bBitStates, sErr and iDErr. The first parameter, zero, was passed to the routine to tell it which chip to read. This is one part of the program that is ready for the multiple chips that are not yet fully implemented. In the variable bBitStates there would be a value that would be determined by what was going into the DS2408. In sErr would be either '0', which means that DMLGetPIOLogicState noticed no problems while executing, or something else pointing to what the error was. In iDall there's the error code returned from the Dallas routines called by DMLGetPIOLogicState. (If sErr is '0', then the contents of iErr are meaningless.) There are two error codes returned for the following reason. Within DMLGetPIOLogicState, there are multiple calls of Dallas routines, any one of which could return, say, error code "27". By looking at sErr and iErr together, you can tell which call resulting in Dallas error "27" was the offending call.

Now we will looking more briefly at DMLWriteToEnabledOutputs. If the following were in your

program.....

```
bVal:=$50;
bMask:=$F0;
DMLWriteToEnabledOutputs(0,bValue,bMask,sErr,iErr);
```

.... the the following would happen:

First, you should think of the numbers in bVal and bMask as binary numbers.... one bit for each of the DS2408's IO lines. (You'll survive, though, even if you're not conversant with binary and hex.) The numbers in bVal and bMask are combined within DMLWriteToEnabledOutputs. If a bit of bMask is 0, that bit is disabled for output operations. Most chips I've worked with are pretty forgiving, but if you accidentally wire an input to a pin that is currently being used as an output, in some circumstances, it can be a Very Bad Thing. Most PIO chips I've dealt with have data direction registers to "set" individual pins as output or input pins. The DS2408 does not use a data direction register, but I'm not sufficiently sure of myself to explain how I think it avoids input-to-output problems. Suffice it to say that DMLWriteToEnabledOutputs will keep things okay, as long as you put the right value in bMask. For the TAI 8558, the "right value" is always $F0. (That's F0, base 16, or hex. I.e. 11110000 binary, or 240 decimal.)

So much for bMask! bVal holds a number saying what you want on the output bits. The four output bits of the TAI 8558 are wired to the four most significant bits of the DS2408's input/output register. In other words, the following binary numbers will turn one and only one of the outputs on: 10000000, 01000000, 00100000, 00010000.

Whew! This low level stuff is so much "fun".

Now we can move up a level.

Within DS032, once the initializations are taken care of, a timer is going to produce a "tick" "tock" situation. About once a second, there will be a tick or a tock. During each "Tick", the program will read the inputs being supplied to the TAI 8558, and make changes, internally, accordingly. I say "internally" because few things in the program will directly access the TAI 8558 to, for instance, start the big bell ringing. During the processing of the inputs, if the program determines that the big bell should be ringing, a flag will be set to record that determination. During the "Tock"s, the program will check the various flags and make a call to DMLWriteToEnabledOutputs with an appropriate number in bVal.

Note I've avoided being too specific about the "right" bit, or whether it needs to be 1 or 0, to make the big bell ring, etc. At this level of the programming, I am not concerned with which bits are connected to what inputs/ outputs. Those considerations are taken care of in the next level of abstraction.

The "Tick Tock" mechanism may seem unnecessarily complex. The reason it exists is that I wanted to reduce the traffic on the MicroLan. This is a Good Idea in general. In addition to the general principle, I have a little problem in my most basic routines. If either is called too quickly after a prior call to either, the system reports errors. I suspect I can get my maximum error-free call frequency down from where it is at the moment. Even if I do, though, there are limits which it will not be possible to overcome. With the "Tick Tock" approach, you can avoid those errors without making the rest of your code dreadfully

complex with things like "write this bit to the device unless we've written to it too recently, in which case make a note to send it when we can".... multiplied by all of the inputs and outputs involved in the system!

I'm nearly ready to move on to the next higher level of abstraction..... but just a little note before I do. Thank heavens I wrote this software in layers.... because everything down to here had to be *re*-written because I'd made a careless assumption. However, almost all of the rest of this needed almost no changes. The "foundations" only needed to provide the basic "read" and "write" mechanisms. The higher levels of the program don't care how they are provided, so they still worked, even though they now work *through* a completely different mechanism!

The next level up.....

What follows in the next section may be a bit of overkill, but if you do the job this way, you'll be "set for life" and future projects with the TAI 8558 will be quite simple.

First we'll create a "virtual" GetPIOLogicState.... it will appear to return information from the DS2408, but what it will in fact do is check a variable which will be holding what was seen the last time the DS2408 was checked.

```
procedure TDS032f1.ReadDS2408(bIdx:byte;
        var bResult:byte;
        var sTKBErr:string;
        var iDalErr:integer);
//"Virtual" read... real read happens when Timer handler deems it should
begin
   bResult:=bPrevResult[0];
   sTKBErr:='frmbuff';//I.e. answer came FRoM BUFFer
   iDalErr:=0;
  end;
```

The second is just a "wrapper" for DMLWriteToEnabledOutputs. It will only be called from within "Tock", but it will look in the global variable passed to it in bVal to see what should be sent to the DS2408. Thus, just change what's in that variable to effect a "virtual write" to the DS2408.

```
procedure TDS032f1.WriteDS2408(bIdx,bVal:byte;
        var sTKBErr:string;
        var iDalErr:integer);
begin
 buReadPIO.enabled:=false;
 bMaskSupplied:=$F0;//specific to DS032
 DMLWriteToEnabledOutputs(bIdx,bVal,bMask[bIdx],sTKBErrReturned,iDalErrReturned);
 bOPStatesBuff[0]:=bVal;
 laDMLTKBErr.caption:=sTKBErrReturned+' (W)';
 if sTKBErrReturned='0' then
  laDMLDalErr.caption:='(meaningless)' // no ; here
 else
```

```
 laDMLDalErr.caption:=inttostr(iDalErrReturned);
end;
```

______________]

Time to go up another level....... The fun begins!

At last! Things get MUCH simpler from now on....

Now we create four more procedures, each of which calls ReadDS2408. Each of them will read one of the input bits. They all return "true" or "false" depending on whether the bit is high or low. (The job could also be done with a single procedure with an extra parameter, but I thought the following was easier to follow!)

The code would be ....

```
procedure TDS032f1.Rds2408i3(bIdx,bMsg:byte;
        var boResult:boolean;
        var sTKBErr:string;
        var iDalErr:integer);
var bResult:byte;
begin
  ReadDS2408(bIdx,bResult,sTKBErr,iDalErr);
  boResult:=(bResult and 8)>0;
  end;

procedure TDS032f1.Rds2408i2(bIdx,bMsg:byte;
        var boResult:boolean;
        var sTKBErr:string;
        var iDalErr:integer);
var bResult:byte;
begin
  ReadDS2408(bIdx,bResult,sTKBErr,iDalErr);
  boResult:=(bResult and 4)>0;
  end;

procedure TDS032f1.Rds2408i1(bIdx,bMsg:byte;
        var boResult:boolean;
        var sTKBErr:string;
        var iDalErr:integer);
var bResult:byte;
begin
  ReadDS2408(bIdx,bResult,sTKBErr,iDalErr);
  boResult:=(bResult and 2)>0;
  end;

procedure TDS032f1.Rds2408i0(bIdx,bMsg:byte;
        var boResult:boolean;
```

```
        var sTKBErr:string;
        var iDalErr:integer);
var bResult:byte;
begin
  ReadDS2408(bIdx,bResult,sTKBErr,iDalErr);
  boResult:=(bResult and 1)>0;
  end;
```

Next, we'll provide ourselves with four procedures which "virtually" turn various output bits on or off. (Again, a "does any one of them" procedure would be possible.) The variable bPrevOPStates is a global variable that continually holds what the state of the outputs is. It is called "previous output states", because it records what we sent to the DS2408 the last time we spoke to it. When the program starts, it is initialized to zero because there is circuitry on the TAI 8558 to make the outputs zero when the device is turned on. The procedures are named to reflect the external devices controlled by the bits each procedure affects. They also change labels displayed on the computer screen. You'll see the words "and" and "or" in the procedures on lines starting bValue:=. In this context, they are "boolean operators" concerned with making specific bits within bValue a 1 or a 0, without changing the other bits.

```
procedure TDS032f1.EnableHiTstat(bOnOff:bit);
var sTKBErr:string;
   iDalErr:integer;
   bValue:byte;
begin
if bOnOff=0 then begin
  laHiTstat.caption:='High Tstat Bypassed';
  bValue:=bOPStatesBuff[0] and (not($80))
  end // no ; here
  else begin
  laHiTstat.caption:='High Tstat Enabled';
  bValue:=bOPStatesBuff[0] or $80;
  end;
bOPStatesBuff[0]:=bValue;
end;

procedure TDS032f1.MainLoudBell(bOnOff:bit);
var sTKBErr:string;
   iDalErr:integer;
   bValue:byte;
begin
if bOnOff=0 then begin
  laMainBell.caption:='Loud Bell Quiet';
  bValue:=bOPStatesBuff[0] and (not($40))
  end // no ; here
  else begin
  laMainBell.caption:='Loud Bell Ringing';
  bValue:=bOPStatesBuff[0] or $40;
```

```
  end;
bOPStatesBuff[0]:=bValue;
end;

procedure TDS032f1.QuietBuzzer(bOnOff:bit);
var sTKBErr:string;
   iDalErr:integer;
   bValue:byte;
begin
if bOnOff=0 then begin
  laBuzzer.caption:='Buzzer silent';
  bValue:=bOPStatesBuff[0] and (not($20))
  end // no ; here
  else begin
  laBuzzer.caption:='Quiet buzzer buzzing';
  bValue:=bOPStatesBuff[0] or $20;
  end;
bOPStatesBuff[0]:=bValue;
end;

procedure TDS032f1.MsgLED(bOnOff:bit);
var sTKBErr:string;
   iDalErr:integer;
   bValue:byte;
begin
if bOnOff=0 then begin
  laMsgLED.caption:='Message LED Off';
  bValue:=bOPStatesBuff[0] and (not($10))
  end // no ; here
  else begin
  laMsgLED.caption:='Message LED On';
  bValue:=bOPStatesBuff[0] or $10;
  end;
bOPStatesBuff[0]:=bValue;
end;
```

 - - - - -

Almost there! the remaining "boring" low level work is to create functions that make calls to DMLGetPIOLogicState and return "true" or "false" to questions like "*Does the PIR "see" an intruder in the house?*".

By the way: a "function" is essentially a new word made up for your programming language by you, for example, consider....

```
if bTmp=5 then showmessage('It is 5');
```

All of the above is legal Delphi. The "bTmp=5" "boils down" to something which is either true or false.

You could make a *function* called "IsBTmp5" as follows:

```
function  IsBTmp5:boolean;
boTmp:boolean;
begin
boTmp:=false;
if bTmp=5 then boTmp:=true;
result:=boTmp;
end;
```

Once you've done that, you can write....

```
if  IsBTmp5 then showmessage('It is 5');
```

.... because IsBTmp5 "boils down" to true or false, just as bTmp=5 did previously. It is a "function".

For our alarm / heating control system, we want 3 functions, as follows. Remember "Alarm should be *set*" is testing whether the "make the alarm system *watch* the house" is turned to the "Do It" position. (If you want the bells *to ring,* you need to “trip” the alarm.) AlarmShouldBeSet is an input. BellShouldRing is an output.

```
function TDS032f1.AlarmShouldBeSet:boolean;
var boRet:boolean;
   sTKBErr:string;
   iDalErr:integer;
begin
Rds2408i1(0,bOPStatesBuff[0],boRet,sTKBErr,iDalErr);
result:= boRet;
end;

function TDS032f1.PIRSeesIntruder:boolean;
var boRet:boolean;
   sTKBErr:string;
   iDalErr:integer;
begin
Rds2408i2(0,bOPStatesBuff[0],boRet,sTKBErr,iDalErr);
result:=boRet;
end;

function TDS032f1.FireDetected:boolean;
var boRet:boolean;
   sTKBErr:string;
   iDalErr:integer;
begin
Rds2408i3(0,bOPStatesBuff[0],boRet,sTKBErr,iDalErr);
result:=not boRet;
//The "not" arises because of the way the switch was wired
```

```
//and the resulting logic state representing
//"Fire". The Rds2408... proc is "answer" neutral... it is
//in little functions like this that the raw results can
//be turned functions with user-friendly names.
end;
```

Hurrah! We at last have our basic toolbox: functions to read each of the inputs, and procedures to make the outputs what we want them. Let the fun programming begin!

We talked about various options for our system. For this part of the discussion, assume the inputs implied by the names of the functions which will be part of the program:

PIRSeesIntruder: False if no intruder seen, true if one is.
FireDetected: False if fire sensors see no fire.
AlarmShouldBeSet: True if the toggle switch requesting the system be armed is on.

Further assume the outputs implied by the names of the following:

EnableHighTstat (when high, alters heating circuit so that heat comes on if temperature is below whatever the "high" thermostat is set to.)
MainLoudBell
QuietBuzzer
MsgLED

We have now reached a point where a simple alarm system would consist of something like the following:

```
repeat
  if AlarmShouldBeSet and PIRSeesIntruder then MainLoudBell(1)
       else MainLoudBell(0)
until false
```

Really! That would almost work. Now do you see why all the trouble was taken to get this far? The "program" would work, under some operating systems. With Windows, a tight infinite loop like that is a Bad Idea, but we can get around that problem....

In order to make our problem manageable, we're going to define some system states.

StateAFamilyHomeAndUp: When the alarm is not armed, this is the state we will normally be in.
StateBAlarmSet: When no one is home, or the family have gone to bed, this is the state we will normally be in.
StateCBellRinging: When the alarm has been tripped, we will be in this state.
StateAtoB: We will be in this state for a short while after a user has asked for the alarm to be armed. While we're in StateAtoB, the alarm won't ring even if PIRSeesIntruder is true. The point of this state is to allow for an exit time.... time for the homeowner to leave the premises after arming the alarm. This

permits us to have the "arm the system" switch in a place "seen" by the PIRs.
StateBtoC: We will be in this state for a short while after the alarm system has been tripped, and it is just about to move into StateCBellRinging. By providing a StateBtoC, we make a place where code could be added for turning off an accidentally tripped alarm before the main bell starts to ring. Of course, it must not be too easy to turn it off! It also makes a place where a "silent alarm" could be "ringing" in advance of the main bell starting to shriek.

There may be others that will emerge as we develop the software.

One little detail I'm going to deal with here: If you modify the code, remember the following: You could have a power failure. When the power comes back on, the “AlarmShouldBeSet” input could be set or not set, depending on how things stood when the power failed. Will your program handle either circumstance properly? (The program presented here does.)

Besides the states, we'll have a few global variables. The first is CurrentState, and in this will be something to show what state we are in at the moment. A code will be used, with "A" standing for "StateAFamilyHomeAndUp", "B" standing for "StateBAlarmSet", "AB" standing for "StateAtoB", etc.

bCountDown is a global variable we will provide in order to keep track of how long we've been in the transition states like StateAtoB.

I've already mentioned another global variable, " bPrevOPStates ", which holds the state of the various outputs.

Within Delphi, and, I presume other languages for Windows, is a component called a timer. Associated with it is an event that is triggered whenever a certain amount of time has passed. You can set the timeout interval to what you want. Our Windows-friendly version of the simple alarm program set out above would just put the following in the Timer Timeout Event Handler, and the timer would be set to time out about once every two seconds. The code should look familiar!

```
if AlarmShouldBeSet and PIRSeesIntruder then MainLoudBell(1)
    else MainLoudBell(0)
```

Moving on. In both of the programs presented to date, a detail was omitted. When the program is started, some initialization code would be executed. A check would be made of AlarmShouldBeSet, and the various outputs would be set accordingly. The system will have been put together so that the outputs for MsgLED, MainLoudBell and QuietBuzzer will not be causing light or noise, but the relevant bits of bPrevOPStates need to be set or cleared as required.

The timer's timeout event handler will have a skeleton like this...

```
begin
if bTickTock=0 then begin //"Tick": read DS2408
bTickTock:=1;//Prepare for next pass.

DMLGetPIOLogicState(0,bResult,sTKBErr,iDalErr);
```

```
bPrevResult[0]:=bResult;

if FireDetected then begin
 MainLoudBell(1);
 QuietBuzzer(1);
 MsgLED(1);
 CurrentState:=ssCs;
 end;
 //Program needs a way to turn fire alarm off, but this
 // is not as simple as it may seem. Saying "bell off when
 // detector no longer sees fire" is not a good idea (fire
 // can burn through wire, and in any case, which state you
 // go to is a complicated question.)

if AlarmShouldBeSet then laArmed.caption:='Alarm asked to watch'
            else laArmed.caption:='Alarm not asked to watch';

if PIRSeesIntruder then laPIRs.caption:='PIR sees someone'
            else laPIRs.caption:='PIR senses nothing';

end //End of "Tick", ReadPIO option of timer

else begin (*"Tock": First check what's needed,
                Second: write to controlled devices*)
bTickTock:=0;
boNPAC:=true;//This flag keeps track of “Have we handled a valid state yet?”

//First, see if we're in state A, and if so deal with it.....
//State A is Family at home and up. (Alarm is set when in bed)
if boNPAC and (CurrentState=ssAs) then begin
  if AlarmShouldBeSet then begin
    CurrentState:=ssABs;
    EnableHiTstat(0);
    QuietBuzzer(0);
    MsgLED(0);
    bCountDown:=7;//Determines time in stateAB
    end;
  boNPAC:=false;
  end;//ssAs

//Next we need the code to see if we're in State B, if so deal with it, then C, then the transitional states.
See the full source code for the details....

end; //"Tock"
```

(A little "extra" work will be necessary, in case you enter the event hander with CurrentState equal to,

say, "B", and during what is done whenever CurrentState is "B" CurrentState is changed to, say, "AB". This isn't hard, but is a detail to take care of.)

We got a bit ahead of ourselves there, but I wanted to cover the “Tick Tock” mechanism. Within the “Tock” phase, we've already seen what has to be done if we are in “State A”. Before that programming was done, a careful analysis had been done, looking at the characteristics of the different states. What does the program need to look for when it is in a given state? What variables need to be set to what upon changing states? What TAI8558 controlled outputs need changing when we change state. I did a diagram (not reproduced here) showing the different states, with arrows marking the changes which had to be provided for. There was also a table setting out all of the variable changes involved.

If you look at the source code, you will see the details of the other states and state changes. Most of what is involved is uncomplicated, but I will explain two things which are a little obscure:

The code which comes into play if a fire is detected is in the “Tick” phase. Not entirely “logical”, but it fit well there

In the three transitional states (AB, BA and BC), there is a call to a procedure called “Signal”, and a variable called bCountDown is in use. A parameter is passed to Signal which is used to determine what the “beep-beep” sound on the small buzzer will sound like: short-on-long-off, long-on-short-off, or balanced on-off. The number in bCountDown decreases by one with each call of “Tock”, and when it gets to zero, the program moves into the state it was headed for.

That pretty well does it for the basic package. Just as an example of the benefits of writing the program well, as we have done, let me show you how easy it would be to add an additional feature.

Suppose you want to get up in a warm house? You can add this feature simply by adding the following in "Tock", just before the WriteDS2408(0,bOPStatesBuff[0],sTKBErr,iDalErr);....

```
(*The next few lines provide for the heating to be
  turned up from just after 6am (0.25 part of a day) to about
  7 (about 0.28) even if alarm is set. The "turn off
  again" is necessary in case everyone is away from
  home!*)

laOverride.caption:='-  -  -  -  -  -  -';
if CurrentState<>ssAs then begin
  dtTmp:=trunc(now);
  if (dtTmp>0.25) and (dtTmp<0.28) then begin
    laOverride.caption:='Thermostat enabled by over-ride';
    EnableHiTstat(1);
    end (*no ; here *)
   else begin
    EnableHiTstat(0);
    end;(*else of dtTmp>0.25...*)
```

```
  end; (*CurrentState<>ssAs*)
```

Similar little embellishments can provide for other things, like.....

Suppose you sometimes forget to switch the alarm on at night? The house's heating to be turned down at 11pm whether the alarm is set or not. Of course, you'll need something else to turn it on again in the morning if the house is in state "A". Normally, the higher heat is turned on when you go from B to A.

Perhaps everyone's at work and school until 4:30pm most days, but usually home not long after. You can add something to give people a warm house to come home to.

The possibilities are endless, but this document must not be! Comments on what you liked or didn't are very welcome. Go to http://sheepdogsoftware.co.uk to contact me, Tom Boyd.

Thank you for reading!

============

Appendix A: An explanation of some terms:

(See the note early in the text regarding references such as "the connectors along the bottom edge of the device.)

1-Wire: See MicroLan.

DS032: The name of the software written to go with the TAI 8558 and the external hardware in order to implement the system this application note describes. It was written in Delphi, but is described in terms which will, I hope, enable programmers of other languages to produce equivalent programs.

DS2408: The Dallas chip at the heart of the TAI 8558.

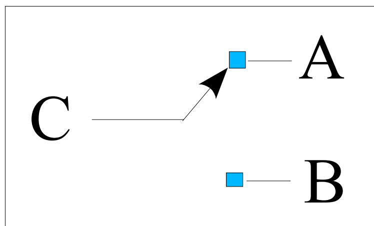
MicroLan: This, and "1-Wire" are registered trademarks of Dallas Semiconductor. 1-Wire devices can be interconnected with nothing more than two wires. Each has a unique serial number, so if you connected, say, 10 temperature sensing 1-Wire chips, the master computer connected to them via the same two wires could tell which chip was reporting any reported temperature. A collection of such chips, and the wires connecting them, may be called a MicroLAN.

Opto-isolator: An opto-isolator is essentially a photo-transistor and one or two LEDs enclosed in a lump of plastic with a bubble of air so that when either LED is lit, the photo-transistor conducts, but if the LEDs are dark, the photo-transistor doesn't conduct. The TAI 8558 uses opto-isolators so that mistakes by a user of the TAI 8558 at worst ruin one fairly easily replaced chip, and also to save users the hassle of providing for 5 volt DC signals when the signals they have are AC, or of a different voltage. On the TAI 8558, there are five or six screw terminal blocks across the top of the unit. Ignore the one on the left for the moment. The next four, two connections each, are the inputs to the units four opto-isolators. Create a current through the two connections of any one of the terminal blocks, and, when you have mastered the software issues involved, the computer running the MicroLan will be able

to "see" that you have created that current.

PIR Unit: The basis of many burglar alarm systems. "PIR" comes from "Passive Infra Red". Unless I am mistaken these devices are a spin-off from military research, and what they learned in making heat seeking missiles has been applied to give us relatively inexpensive units that "see" the heat given off by humans. Not only do PIRs look for heat, they look for moving sources of heat. When the unit sees no intruder, a "switch" inside the device remains closed. When the unit sees moving heat, the "switch" opens.

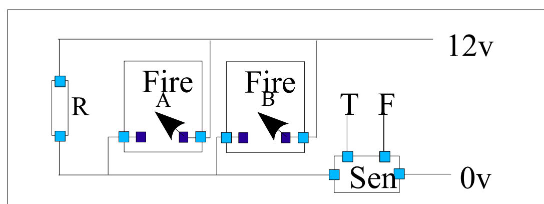
Relay: This is a "switch" which can be turned on or off electrically. The relays in the TAI 8558 are switched on or off by messages sent to the unit using the 1-Wire protocol. I.e., when you've mastered the software side of this project, you'll see that you can send a command saying "turn relay K1 on". As a user, you have three places you connect wires to the relays. The connections are via the screw terminals along the bottom edge of the board. Inside each relay, as I said, is a switch. It is a single pole, double throw switch, i.e.:



The letters A, B, C represent the three connections which you can make to the relay. "C" is for "common", because it is "common" to the circuit whether the switch is up or down. The switch is changed from A to B by means of a small electromagnet. When the magnet is off, the switch is up, when it is on, the magnet is down. "A", in that scenario, would usually be referred to as the "NC" contact, because the connection between C and A is "Normally" Closed, i.e. "made" when the relay is off, the so-called "normal" state. Contact B would usually be referred to as the "NO" contact, because the connection between C and B is "Normally" Open. On the TAI 8558, the central connector in each screw terminal (across the bottom edge of the board) is the Common connector. The left connector goes to the NC pole of the switch.

"Set the alarm": make the system enter the state in which it will trip the alarm if it "sees" something wrong. ("Unset the alarm" to mean return it to the state in which people can move about the house without tripping the alarm.)

Supervised circuit: At the price of some extra complexity, a string of fire detectors can be of the "close on detecting fire" type, and still protected from the failure which would result if the wire had been broken, and no one had noticed. The circuit would be a little different from the one shown in the main section of this application note.



Supervised circuit. "R" is a resistor, wired to the far extremity of the circuit. "Sen" is the sensing circuit. When no current is flowing through it, the "T" (trouble) output would go high. When a little current is flowing through "Sen", neither output is high. When a large current is flowing through "Sen", the "F" (fire) output goes high.

"Trip the alarm": set the loud "something's wrong" bell wailing.

"Unset the alarm": return it to the state in which people can move about the house without tripping the

alarm.. ("Set the alarm": make the system enter the state in which it will trip the alarm if it "sees" something wrong)

==============

Appendix B: Powering the TAI 8558

See the section called **The connections at "B"**, earlier in this document (part of **Connecting to the TAI 8558**) for the fine details on this point, but, in summary:

a) I *think* that the TAI 8558 can be operated without any power, other than that provided by the two basic 1-Wire wires, as long as you only want to use it to read inputs, and as long as you remove the jumper. However, as this seems unlikely to be very useful, I'm not checking the details too carefully. Removing the jumper will disable the input-indicating LEDs, not that I can see why you'd want to do that!

b) If you have a source of regulated 5v DC, you can connect that to the TAI 8558 via the same connector used for the 1-Wire wires, whether you are using RJ-12's or the three connector screw terminal which can be fitted in their place. If you use this option, remember that if you try to send 5v to the device over a long wire, it may not be sufficiently close to 5v by the time it arrives.

c) You can supply an unregulated power source, AC or DC, via the two connector screw terminal in the upper left corner of the board. If you are supplying DC, it doesn't matter which way 'round you connect the wires because of the bridge rectifier in the TAI 8558. The nominal requirement is 12v DC, but as long as you get a good 5v across the little "can"-type capacitor in the lower left corner of the board, lower voltages may be used. Be sure the 5v is still 5v when all of the LEDs are glowing. If you supply a too-high voltage, the heatsink on the voltage regulator won't be sufficient. (The voltage regulator is the large three pin rectangular device in the middle of the left edge of the board.) I was running the device off of a fairly basic "power brick" which gave between 6.9v and 8.5v depending on how many LEDs and relays were on. In this environment, the TAI 8558 ran just fine. By the way: When that power brick was set to deliver "12v", it actually delivered rather more, perhaps too much.

I'd be interested to hear your ideas! Tom Boyd, http:/sheepdogsoftware.co.uk

==============

Appendix C: Other uses for inputs and outputs

Here are a few other ideas for using inputs and outputs if you buy a second TAI 8558, or if you don't implement all of the ideas above.

It would be quite simple to turn a light or lights on and off with the system. Remember, though, that when dealing with household electricity, extra precautions need to be taken. Because a computer is

involved, the timing of the lights' ons and offs can be more sophisticated than a time switch would permit.

Even with clever on / off regimes, automatic lights are probably not going to fool many people watching an empty house with a view to burglary. If blinds (curtains) are being opened and shut at dusk and dawn, burglars may be fooled. Alternatively, you may want to control blinds to affect the temperature in a greenhouse. What you need to do is fit blinds which can be opened or closed by pulling on a loop of cord. Wrap that cord around the shaft of a DC motor. When it turns one way, the blinds open, the other way, they close. Next fit two switches: one turns the motor off when the blinds are fully open, the other when they are fully closed. The motor can be made to turn either clockwise or the other way by a simple wiring of the power through two single pole, double throw switches, e.g. two of the relays on the TAI 8558. ***Always be careful when driving DC motors:*** The voltages which are induced when power is removed can fry delicate digital electronics. A diode is essential, and I'd use a completely separate power supply.

There are sundry ways to arrange for automatic watering of plants. Because of the damage that water can cause, I'm am particularly careful to try to devise fail-safe designs. Most use windscreen washer pumps to move the water.... they are easily available and run on 12DC electricity. They require the same care mentioned in the previous paragraph.

Appendix D: Basic 1-Wire communications:

To use the chip at the heart of the TAI 8558, a DS2408, you need a master computer connected to the chip over a 1-Wire MicroLan. I'll sketch the process below. For details, using Delphi (but from that, you should be able to translate to the language you like) see http://www.arunet.co.uk/tkboyd/e1didx.htm.

Your program will "talk" to some .dlls in your computer. They talk to the chip.

First you need to open communications with the .dlls. This is called Establishing a Session. Once the session is established, you can leave it in place for as long as the application runs. You should close the session as the application shuts down. It is quite like establishing a file handle to do disk based I/O.

Then, each time you want to talk to a chip on the MicroLan, you follow the following sequence.

1) "Say" to the MicroLan "I'm sending a message to chip number 2348266...., the rest of you ignore what follows". (Each chip has a unique numerical id.) The application can search the MicroLan and compile a list of chips present, or something like an ini file can be used to establish what chips are on the MicroLan.

2) For the DS2408, the next thing to do is either to send a byte to it, or to read from the DS2408 to learn what the inputs are seeing. My diagram concerning the TAI 8558's various LEDs and connectors has the inputs and outputs labeled to correspond to the assignment of bits to ports. For example, the least significant bit of the byte (bit 0) will tell you the state of the input on the channel connecting to the screw terminal in the upper right of the board. Weird (to me, anyway!): The DS2408 doesn't need a data

direction register.

3) Once you've had your "conversation" with "the chip of the moment", you send a message over the MicroLan reseting the other chips, so that they are listening out for a call as in step 1 above, which you can issue when you are ready to "talk" to the next chip.

==

Comments?? Please send any to: Tom Boyd, http:/sheepdogsoftware.co.uk