DOCUMENTATION FOR THE 'GBSCommsMod.AXS' MODULE

1. 'GBSCommsMod.AXS' MODULE DESCRIPTION

Instantiating a module in an include file as illustrated here, allows for easy inclusion of the module in your project, because all required variables (variables that are to be sent to the module as parameters) are declared in the include file and any custom code associated with the module that is being instantiated can be placed in that include file. That makes the file structures logical and it makes it easier to find code associated with the module. So, in this example, a #INCLUDE statement is entered in the Main program for the file called 'Green Box Systems Wired Initialisation.AXI' and then the 'GBSCommsMod' module is instantiated in that include file. All variables, devices and virtual devices pertaining to the module are declared in the include file.

The provided include file also shows some example code, e.g. how to switch lights on and off or dim them, and how to use the feedback from the 'GBSCommsMod.AXS' variables.

This module has been provided as a communications interface between an AMX controller and a Green Box Systems (GBS) Wired lighting system, using either an IP connection or a direct RS485 connection to the lighting system. The GBS wired lighting system is based on an RS485 bus where multiple GBS dimming, switching and other devices can be connected together and where each RS485 address on the RS485 network needs to be unique. The GBS wired lighting system also has the option of an IP interface which is provided for scalability (multiple GBS RS485 networks can be connected to each other via a TCP/IP backbone). When ordering a GBS system, the integrator should have an IP interface for set up of the GBS system (even if the IP interface is not used permanently in the installation). For example, it may be necessary to change RS485 bus addresses of the devices that are connected to a GBS wired lighting system. It should be noted that each GBS device has a unique address in the sense of two parameters, a subnet ID (that is which RS485 network it belongs to, not be confused with the concept of a TCP/IP subnet), and a device ID on that RS485 network. RS485 networks can only connect to each other via a GBS IP interface and a TCP/IP backbone. This AMX module only addresses subnet 1. So, for each RS485 network to be controlled by this AMX module, a new instance of this module needs to be used and the interface to each of those GBS RS485 networks can be achieved either by a different TCP/IP socket or a different RS485 port on the AMX system. When using this AMX module each GBS device should be addressed using subnet 1 and then the unique device ID of that device. Most programmers will not necessarily get involved with this addressing, because Peripheral Vision intends to supply the GBS devices already addressed with unique device ID’s (all on subnet 1). A Windows application is available and can be supplied to the programmer on request, which can be used to change the subnet ID’s and device addresses of the devices on the GBS RS485 network. However as stated, this should mostly not be necessary.

This module uses multiple virtual devices to communicate with the multiple physical devices on the GBS RS485 network. So, for example if there are three GBS devices on an RS485 bus which AMX will interface with and control, three unique virtual device ID’s will need to be defined in the 'Green Box Systems Wired Initialisation.AXI' file. These virtual devices are then combined into an array of virtual devices (in this example code, an array of two virtual devices is declared with the statement:

VOLATILE DEV vGBS\_DEV\_ARRAY[] = {vdvGBS\_4\_DIMMER\_1,vdvGBS\_4\_DIMMER\_2};

Two other arrays also need to be defined to instantiate this module (as shown in the example code provided in the 'Green Box Systems Wired Initialisation.AXI' include file), in the following manner:

VOLATILE INTEGER vGBS\_DEVICE\_ID\_ARRAY[] = {2,6};

VOLATILE INTEGER GBS\_INDIVIDUAL\_DEBUGS[] = {0,0};

The meaning of these variables is described later in this document, but it is important at this point to understand that there is significance in the array element references. For example, the first array element in the array vGBS\_DEV\_ARRAY (which is vdvGBS\_4\_DIMMER\_1) is the first virtual device, and represents the first RS485 device on the GBS RS485 network. At the same time, the first array element in the array vGBS\_DEVICE\_ID\_ARRAY (which is 2), is the device ID of that first GBS device on the GBS RS485 network which (as stated above) is represented by the first virtual device in the array vGBS\_DEV\_ARRAY. The same applies for the array GBS\_INDIVIDUAL\_DEBUGS (the first element is associated with the first element of the other two arrays). Also, the same applies for the second elements in all three arrays (and the third or fourth or more elements, if they exist).

1. HOW TO INSTANTIATE AND USE THE GBSCommsMod MODULE
   1. Devices:

As stated previously, it is possible to connect to, and control a GBS over IP or RS485. If you connect over IP, the RS485 mode needs to be disabled (and the same vice versa).

* + 1. The IP method of connection is achieved by the following criteria:
       1. The device dvGBS\_IP\_TX\_DEVICE (or whatever you prefer to call it) needs to have a valid definition. This device is the IP socket that will be used to TRANSMIT to the GBS system. For this device to be a valid IP socket, it should start with 0 (which means it is an IP socket off the master).
       2. The device dvGBS\_IP\_RX\_DEVICE (or whatever you prefer to call it) needs to have a valid definition. This device is the IP socket that will be used to RECEIVE comms from the GBS system. The reason why two separate IP sockets are used is because the GBS system replies on a multicast IP address (IP Address '192.168.255.255', which is the multicast IP address that is defined for the '192.168.0.0' IP network). Because I am using '192.168.255.255' as the multicast IP address to listen on, you should put your AMX and GBS system on any IP address starting with '192.168'. For this device to be a valid IP socket, it should start with 0 (which means it is an IP socket off the master).
       3. The parameter GBS\_TX\_IP\_ADDRESS (or whatever you call it) should contain a string version of the IP address of the IP interface of the GBS wired system. An example of the required statement would something like this: GBS\_ TX\_IP\_ADDRESS = ‘192.168.1.111’;).
       4. The parameter GBS\_IP\_PORT should be declared and initialised with the TCP/IP software port being used for the TCP/IP socket to the IP interface of the GBS wired system. The default TCP/IP software port used on GBS wired lighting systems is port 6000, and should only change from this if it is specifically changed by the programmer to an alternative.
       5. For each physical GBS RS485 device, a device such as vdvGBS\_4\_DIMMER\_1 (or whatever you prefer to call it) should have a valid definition. These are virtual devices, so they should be defined in the correct address range for virtual devices.
       6. So, as shown in this example program, the following devices have been defined correctly for the IP method of connection to succeed (Obviously, your program should not contain any other device or virtual device that clashes with these devices):

dvGBS\_IP\_TX\_DEVICE = 0:13:0

dvGBS\_IP\_RX\_DEVICE = 0:14:0

vdvGBS\_4\_DIMMER\_1= 33001:1:0

vdvGBS\_4\_DIMMER\_2 = 33002:1:0

* + 1. The RS485 method of connection is achieved by the following criteria:
       1. The device dvGBS\_IP\_TX\_DEVICE (or whatever you prefer to call it, in fact it would probably be renamed to something such as dvRS485\_DEVICE) needs to have a valid definition. This device is the RS485 port that will be used to TRANSMIT AND RECEIVE FROM the GBS system. For this device to be a valid RS485 port, it should NOT start with 0 (which means it is an AMX physical port).
       2. The device dvGBS\_IP\_RX\_DEVICE (or whatever you prefer to call it, in fact it would probably be renamed to something like dvUNUSED\_GBS\_IP\_RX\_DEVICE to show that it is not being used) should be defined with the device ID 0:0:0 (which means it will not be used). It will not be used because the RS485 connection takes care of all transmitted and received data.
       3. The parameter GBS\_TX\_IP\_ADDRESS should be an empty string (or whatever you call it, in fact it would probably be named and initialised like this: GBS\_UNUSED\_TX\_IP\_ADDRESS = ‘’;).
       4. The parameter GBS\_IP\_PORT (or whatever you prefer to call it, in fact it would probably be renamed to something like GBS\_UNUSED\_IP\_PORT to show that it is not being used) should be declared (and initialised as 0 with a statement such as: GBS\_UNUSED\_IP\_PORT = 0;).
       5. For each physical GBS RS485 device, a device such as vdvGBS\_4\_DIMMER\_1 (or whatever you prefer to call it) should have a valid definition. These are virtual devices, so they should be defined in the correct address range for virtual devices.
       6. So, as shown in this example program, the following devices have been defined correctly for the RS485 method of connection to succeed (Obviously, your program should not contain any other device or virtual device that clashes with these devices):

dvRS485\_DEVICE = 5001:1:0

dvUNUSED\_GBS\_IP\_RX\_DEVICE = 0:0:0

vdvGBS\_4\_DIMMER\_1 = 33001:1:0

vdvGBS\_4\_DIMMER\_2 = 33002:1:0

* 1. Variables:

The following variables need to be defined so they can be used as parameters to be sent to the module. You can call them whatever you like, as long as you insert them in the correct place in the module instantiation statement.

VOLATILE DEV vGBS\_DEV\_ARRAY[] = {vdvGBS\_4\_DIMMER\_1,vdvGBS\_4\_DIMMER\_2};

VOLATILE INTEGER vGBS\_DEVICE\_ID\_ARRAY[] = {2,6};

VOLATILE INTEGER GBS\_INDIVIDUAL\_DEBUGS[] = {0,0};

VOLATILE INTEGER GBS\_NODE\_COUNT = 2;

VOLATILE INTEGER GBS\_DEBUG = 0;

VOLATILE INTEGER GBS\_LIGHTING\_LEVELS[64][48];

VOLATILE INTEGER GBS\_SWITCH\_STATES[64][48];

VOLATILE INTEGER GBS\_MODULE\_VERSION;

VOLATILE INTEGER GBS\_IP\_PORT = 6000;

VOLATILE CHAR GBS\_TX\_IP\_ADDRESS[] = '192.168.1.8';

* + 1. DESCRIPTION OF EACH VARIABLE:
       1. The array variable vGBS\_DEV\_ARRAY[] contains virtual devices, which are used to communicate with the physical devices on the GBS bus. So, if you have ten devices on the GBS bus, you will have 10 elements in the array vGBS\_DEV\_ARRAY[], which are virtual devices representing each device on the GBS bus. The GBS devices will be controlled by the sending of COMMAND's (using the correct syntax) to the virtual devices in this array. For example, if you want to switch a dimmer on the GBS bus to 100% brightness, and that dimmer is the first element in the array of virtual devices, you would send a command to that first virtual device like this:

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[1]:',ITOA(100)"

You could also send the COMMAND to the virtual device itself (and not the element of the array of virtual devices), such as the following example (using the virtual device that has been declared in this program):

SEND\_COMMAND vdvGBS\_4\_DIMMER\_1,"'L:[1]:',ITOA(100)"

To clarify, in this program, there is only TWO elements in the array of virtual devices (vdvGBS\_4\_DIMMER\_1,vdvGBS\_4\_DIMMER\_2).

* + - 1. vGBS\_DEVICE\_ID\_ARRAY[] is an array variable which contains all the GBS device ID's of the devices on the GBS bus. The array indexes in this array correspond with the array indexes of the vGBS\_DEV\_ARRAY[] array variable. So, in other words, the first device ID in the array variable vGBS\_DEVICE\_ID\_ARRAY[] is the GBS device ID of that device on the GBS bus, and corresponds to the first virtual device in the array variable vGBS\_DEV\_ARRAY[]. So, if you send a COMMAND to the first virtual device in the array variable vGBS\_DEV\_ARRAY[], it will be forwarded by the module to the GBS address stored in the first element of the array variable vGBS\_DEVICE\_ID\_ARRAY[].
      2. GBS\_INDIVIDUAL\_DEBUGS[] is an array of debug variables which can individually be turned on and off, which enables or disables diagnostic messages in the Diagnostics window for specific addresses on the GBS bus. This is provided to reduce the communications from ALL the devices on the GBS bus (there are a lot of spontaneous background messages on the bus, which can cloud the communications that the programmer is trying to see. The array elements relate to the array elements of the virtual device addresses in the array variable vGBS\_DEV\_ARRAY[].
      3. GBS\_NODE\_COUNT contains the number of GBS devices on the GBS RS485 network and should be declared and initialised with the number of RS485 devices on the GBS RS485 network.
      4. GBS\_DEBUG is a variable which turns on and off certain diagnostic messages in the Diagnostics window. This debug variable turns off those diagnostics messages for all GBS devices on the GBS bus.
      5. GBS\_LIGHTING\_LEVELS[64][48] is a two-dimensional array which the module will store the latest lighting levels in. The first dimension of the array represents the index of each device (in the array variables vGBS\_DEV\_ARRAY[] or vGBS\_DEVICE\_ID\_ARRAY[], depending on how you look at it), while the second dimension represents the channel number in that device.
      6. GBS\_SWITCH\_STATES[64][48] is a variable which does not need to be considered for the purposes of systems where AMX will control all the lights, and there are no GBS user interfaces involved. However, this variable should still be declared and used as the relevant parameter for the module instantiation command.
      7. GBS\_MODULE\_VERSION is a variable that stores the version number of the module (only whole version numbers are supported).
      8. GBS\_IP\_PORT is a variable that should be edited to set the IP port that the AMX system will connect to the GBS IP port. This port can be changed, but default it should be 6000, so this does not need to be edited in the normal situation.
      9. GBS\_TX\_IP\_ADDRESS[] is an array of char variable (STRING) which should be edited to reflect the IP address of the IP interface that AMX will use to connect to the GBS system.

1. MODULE IMPLEMENTION PROCESS:

The module is implemented simply by the statement:

DEFINE\_MODULE 'GBSCommMod' IP\_EXAMPLE\_INSTANCE (vGBS\_DEV\_ARRAY, vGBS\_DEVICE\_ID\_ARRAY, GBS\_NODE\_COUNT, dvGBS\_IP\_TX\_DEVICE, dvGBS\_IP\_RX\_DEVICE, GBS\_TX\_IP\_ADDRESS, GBS\_IP\_PORT, GBS\_LIGHTING\_LEVELS, GBS\_SWITCH\_STATES, GBS\_MODULE\_VERSION, GBS\_DEBUG, GBS\_INDIVIDUAL\_DEBUGS)

The parameters being sent to the module need to be populated in the statement as shown. This example is for an IP connection (it can be seen because of the parameter names), but an implementation of an RS485 connection method would look something like this:

DEFINE\_MODULE 'GBSCommMod' RS485\_EXAMPLE\_INSTANCE (vGBS\_DEV\_ARRAY, vGBS\_DEVICE\_ID\_ARRAY, GBS\_NODE\_COUNT, dvRS485\_DEVICE, dvUNUSED\_GBS\_IP\_RX\_DEVICE, GBS\_UNUSED\_TX\_IP\_ADDRESS, GBS\_UNUSED\_IP\_PORT, GBS\_LIGHTING\_LEVELS, GBS\_SWITCH\_STATES, GBS\_MODULE\_VERSION, GBS\_DEBUG, GBS\_INDIVIDUAL\_DEBUGS)

1. USING THE MODULE TO CONTROL LIGHTS:

COMMAND’s are sent to individual virtual devices to effect lighting controls on the intended RS485 device that corresponds to the virtual device that the COMMAND was sent to. So, for example, if a lighting control is to be effected on the first GBS RS485 device, a COMMAND will be sent to the first virtual device in the array of virtual devices that were declared in the process of instantiating the module.

For switching that light zone off, the format of the command sent to the virtual device is as follows:

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[1]:0'"

Notice that the COMMAND is sent to the first element of the array of virtual devices. The command is interpreted by the module as an instruction to do a lighting control on channel 1 of the GBS RS485 device represented by the virtual device stored in array element 1 of the array containing all the virtual devices; the lighting level should be set to 0, which will turn that light zone off. For clarification purposes, the COMMAND received by the module will be interpreted into the correct RS485 communication on the RS485 bus and will be sent to the first RS485 device on the bus (or the RS485 device ID 2 (according to the first element defined in the array vGBS\_DEV\_ARRAY in this example program).

Another example of a command to be sent to a virtual device to switch channel 3 to 100% is as follows:

SEND\_COMMAND vGBS\_DEV\_ARRAY[2],"'L:[3]:100'"

In this case, the command is being sent to the second virtual device, so it will be forwarded in the correct format to the second GBS RS485 device on the RS485 bus (represented by the second virtual device in the array of virtual devices), and will control channel 3 of that GBS device to go to a 100% light level. Once again, the RS485 communication is forwarded by the module to the GBS RS485 bus and sent to the GBS RS485 device ID 6 (according to the second element defined in the array vGBS\_DEV\_ARRAY in this example program).

Ramping of lights can be achieved, but the module does not directly support this. Additoinal code should be used to achieve this. It is recommended that this should be used minimally because it keeps the RS485 bus very busy. So, a correct use of ramping of lights would be to create lighting presets (a BUTTON\_EVENT for each of lights OFF, LOW, MEDIUM and ON), and then the lights should be nudged up or down from there. Example code can be provided for this functionality, and the way the module has been written, this will be achieved with full feedback.

1. USING THE MODULE TO GET FEEDBACK ABOUT THE CURRENT STATE OF THE LIGHTS:

The two dimensional array variable GBS\_LIGHTING\_LEVELS is used by the module to store the latest lighting levels of the GBS RS485 devices’ lighting levels. The first dimension of this array represents the index of the device (in the array variables vGBS\_DEV\_ARRAY and vGBS\_DEVICE\_ID\_ARRAY), while the second dimension of this variable represents the channel number of that GBS RS485 device. So, to take the two examples of lighting controls that were used above, the result would be that the variable GBS\_LIGHTING\_LEVELS[1][1] will be equal to 0, and GBS\_LIGHTING\_LEVELS[2][3] will be equal to 100. Using the provided example program and parameters, for the first example in point 4 above the GBS RS485 device with an RS485 device ID of 2 will have its channel 1 at a lighting level of 0, and for the second example, the GBS RS485 device with an RS485 device ID of 6 will have its channel 3 at a lighting level of 100.

Notice that the two dimensional array GBS\_LIGHTING\_LEVELS in this example is defined with a second dimension size of 48, which means that the module can cater for up to 48 channels in any GBS RS485 device. This is only really relevant when the GBS DALI interface is used, where up to 48 channels can be defined. But for the most part, when not using DALI interfaces, you could define this variable with the second dimension size according to the GBS RS485 device with the largest number of channels (which would typically be 4, 6 or 8 channels, depending on what GBS RS485 hardware you have installed in your system).

An alternative way to get feedback from the module is to monitor level events on the virtual device representing each GBS RS485 device. For channel 1 of the GBS RS485 device, a level event will be received on level code 1 of the relevant virtual device. Similarly, for channel 2 of the GBS RS485 device, a level event will be received on level code 2 of the relevant virtual device.

1. EXAMPLE CODE IN THE SUPPLIED AXI FILE

The following code is provided as examples of lighting controls and how the lighting level feedback could be used for those lighting controls. This code is remarked out so as not to interfere with the programmer’s program.

These are the lighting controls, which switches the lights of channel 1, 2, 3 and 4 off for the first BUTTON\_EVENT, and fully on for the second BUTTON\_EVENT.

BUTTON\_EVENT[dTP,1]

{

PUSH:

{

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[1]:',ITOA(0)"

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[2]:',ITOA(0)"

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[3]:',ITOA(0)"

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[4]:',ITOA(0)"

}

}

BUTTON\_EVENT[dTP,2]

{

PUSH:

{

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[1]:',ITOA(100)"

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[2]:',ITOA(100)"

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[3]:',ITOA(100)"

SEND\_COMMAND vGBS\_DEV\_ARRAY[1],"'L:[4]:',ITOA(100)"

}

}

The example feedback code is as follows:

WAIT 5

{

[dTP,1] = ((GBS\_LIGHTING\_LEVELS[1][1] == 0) AND

(GBS\_LIGHTING\_LEVELS[1][2] == 0) AND

(GBS\_LIGHTING\_LEVELS[1][3] == 0) AND

(GBS\_LIGHTING\_LEVELS[1][4] == 0))

[dTP,2] = ((GBS\_LIGHTING\_LEVELS[1][1] == 100) AND

(GBS\_LIGHTING\_LEVELS[1][2] == 100) AND

(GBS\_LIGHTING\_LEVELS[1][3] == 100) AND

(GBS\_LIGHTING\_LEVELS[1][4] == 100))

}

Further example code is provided in the AXI file which processes the LEVEL\_EVENTS provided by the module on the virtual devices (in this case on the first virtual device). These level events keeps the program more event driven (which reduces the unnecessary processor load of running MAINLINE code), which is a good practise.

LEVEL\_EVENT[vdvGBS\_4\_DIMMER\_1,1]

{

iDimmer1Channel1Level = level.value;

}

LEVEL\_EVENT[vdvGBS\_4\_DIMMER\_1,2]

{

iDimmer1Channel2Level = level.value;

}

LEVEL\_EVENT[vdvGBS\_4\_DIMMER\_1,3]

{

iDimmer1Channel3Level = level.value;

}

LEVEL\_EVENT[vdvGBS\_4\_DIMMER\_1,4]

{

iDimmer1Channel4Level = level.value;

}