Manhattan 2 Smart Building Development Initiative

By Glenn Weinreb, CTO, Manhattan 2, Printed Nov 6, 2020

1 Smart Building Interconnection-Standards Development

Manhattan 2 (Ma2) intends to develop electrical, mechanical, and communications standards that define how devices interconnect within the building of the future. Devices include motors that control thermal covers over physical wall windows, motors that control curtains and blinds, fans in ducts, dampers in ducts, lights, occupancy sensors, washing machines, ovens, refrigerators, dish washers, HVAC systems, pumps that control 58°F ground source water, thermal storage water, valves on room water-filled radiators, etc.

This initiative involves developing a 2-wire communication standards between multiple devices that provides the following features: supports CANbus communication between ~$1 microprocessors, no damage upon accidental short to power wires, devices use little power when not in use (sleep), wiring supports tree topology (daisy chain not required), hot socket compatible (no damage when attach wires with power on), ≥ 99.999% reliable (not wireless), and transceivers consume ≤ ~10mW of power when signaling (as opposed to ~10x more utilized by RS-485 or 120Ω CANbus).

We are calling this new network “BuildingBus™”, for lack of a better term, and there are two versions. *BuildingBus 48V* caters to lower power and lower voltage whereas *BuildingBus AC* caters to higher power and higher voltage. *BuildingBus 48V* routes 48VDC power to devices with ~200W/network; whereas *BuildingBus AC* routes 110/220VAC power with ~2,000W/network to devices. Fans, industrial lighting, and motors that move heavy windows require 110/220VAC; whereas many other devices are 48V capable. 110/220VAC cable is bulky and needs to conform to high voltage building codes (i.e. conduit more likely), whereas 48VDC cable is lighter and satisfies low voltage building requirements. For details, search for "BuildingBus Development Initiative", .

We make use of existing standards whenever possible, and propose new standards as needed.

Researchers do *not* necessarily design products to be manufactured and sold. Instead, they propose interconnection standards, and prototypes that demonstrate those standards. These are then provided to standards bodies (e.g. IEEE), which modify as desired, and establish plug-and-play standardization.

All materials produced by researchers are given away for free, to encourage utilization by standards bodies, to reduce CO_2 emissions. This includes mechanical drawings, electrical schematics and software source code.

Ma2 is also developing standards that define how solar material attaches directly to building surfaces, such as plywood. The proposed solar material is ~1.5cm thick and contains embedded electronics that perform power conversion. This material can be applied to both roof and wall surfaces, edge-to-edge.

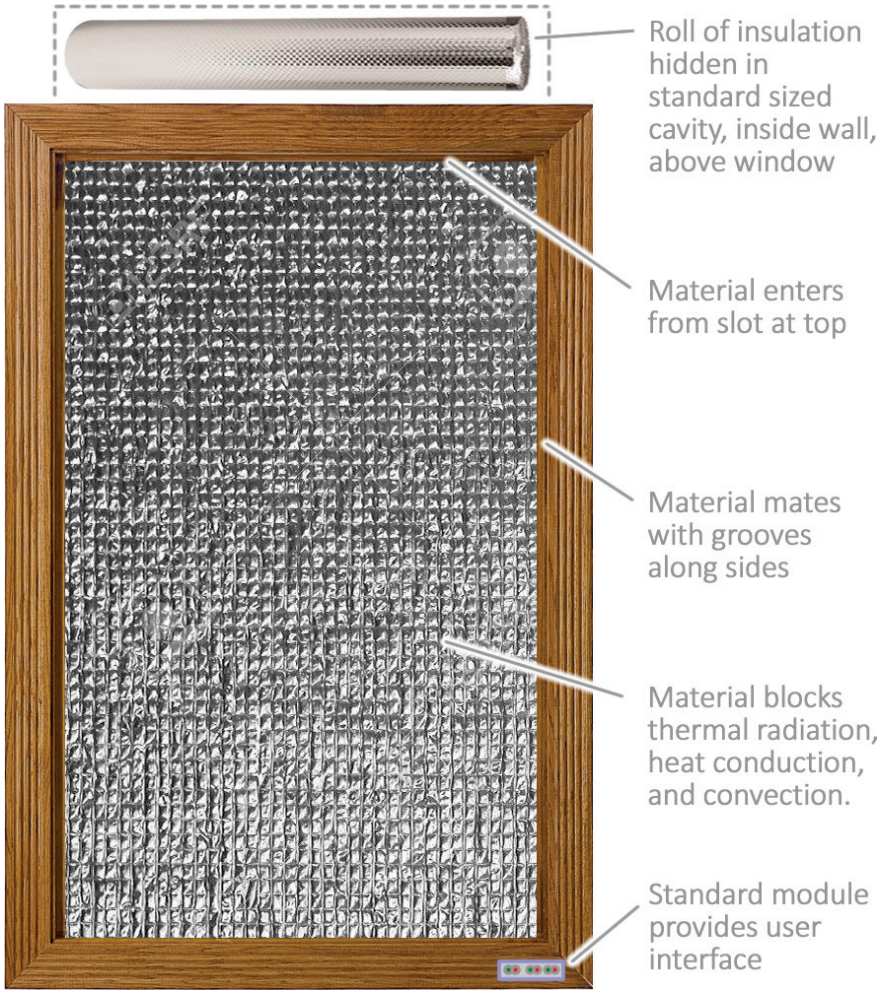
Prototypes develop by researchers use the same processor, and utilize common code. This helps researchers move quickly. Cost reduction is a later step, done by industry, after standards are finalized. The Xmc4200 processor, for example, supports almost all devices. A prototype that moves a window thermal cover, and a prototype of the DC-DC converter embedded in solar material, can both be implemented with this one processor, for example. It provides: 16x 12bit a/d channels, 2x 12bit d/a channels, analog comparators, 2 CANbus channels, counter/timers, 256KBFlash, and 40 KB Ram. All within one tiny package.

See Also:

- Smart Building R&D Initiative
- Rollable Solar R&D Initiative
- Fan and Damper R&D Initiative
- BuildingBus Development Guide. For free and open source code, click here.
- IoT Reference Guide

2 What is an Active Window?

There are two types of physical wall windows, Active and Passive. Active is when a window is powered by electronics, contains a microprocessor, and is connected to a building's network. Active is capable of reducing energy consumption via a variety of methods, one of which is deployment of a motorized thermal cover and thermally turning the window into a wall. Active barely exists due to several issues that we intend to address.



Active supports *Motors*:

- Rolled motorized thermal cover mates with rails at window sides and provides additional thermal insulation.
- Motorized 1 to 2 inch thick solid foam thermal barrier embedded in wall, above and/or below window, deploys to cover window. Many windows provide R3 amount of insulation. Solid foam might provide R7 per inch. Two inches is R14 (2x7). R3+R14 = R17. R17 vs R3 means that

conducted heat loss is reduced 5-fold. Also one can reduce thermal radiation heat loss with "reflective" coatings.

- Motors also move rolled blinds, venetian blinds, and curtains.

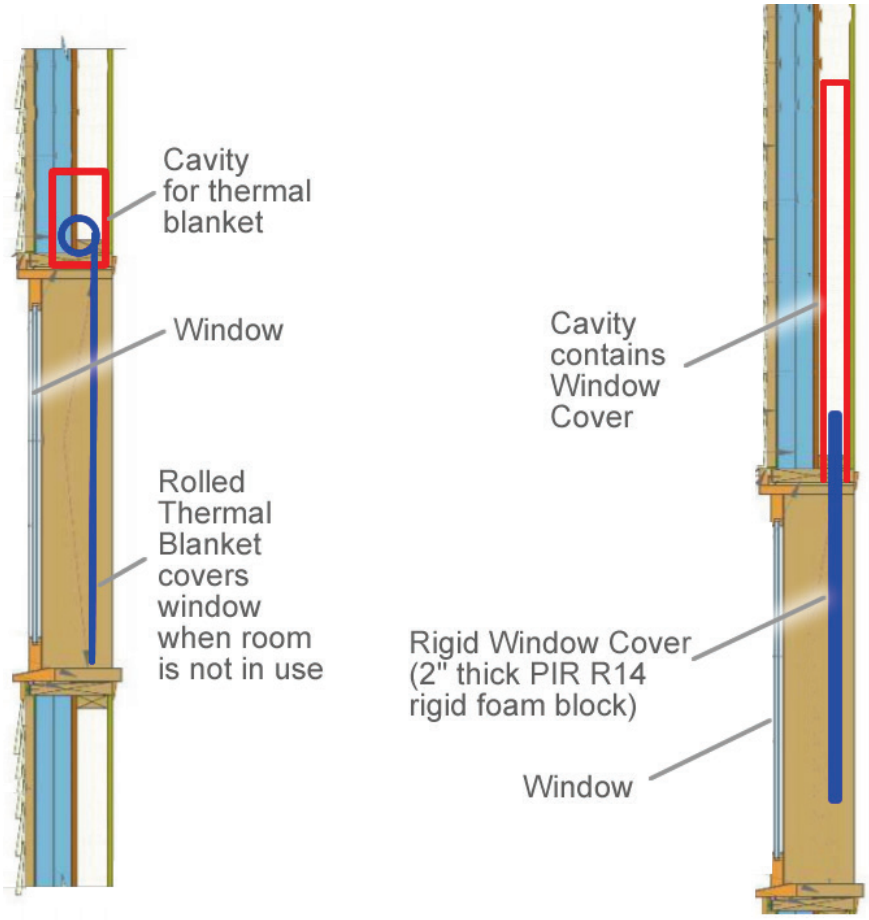
Active Windows can *Sense*:

- Measure indoor and outdoor temperature, humidity and pressure.
- Measure outside sunlight

Active can respond to conditions, to reduce energy consumption:

- If indoor heater or air conditioner is on, room is vacant, and we do not want sun to warm the room; then we want as much thermal insulation as possible via thermal covers, curtains and blinds.
- If pressure is greater inside than out (e.g. due to wind direction) and we want air to flow out, then we crack window. We do the same if pressure is greater outside than in, and we want air in.
- Cracking window also helps to control humidity and CO_2 (i.e. ventilation).

Products that involve electrified gadgetry for windows exist, yet do not sell well for several reasons:



- There is no standard way to attach to a building, which means one must use a proprietary system that involves costs from components, installation and training.
- If one embeds electronics in the wall and it is co-located with 110/220VAC power, and electrician touches power wire to a data wire (e.g. RS-485 or DALI), then physical damage occurs to all devices on network.
- If an embedded motor or gearbox fails 10 to 20 years after construction and is no longer manufactured, then building slowly degrades. People who fund the construction of buildings avoid risk. They will not construct a building that is not expected to function well over 50 to 100 years. Also, buyers do not want a building that degrades over time, since they lose value.

In this document, the term "search" refers to the ~130 page Ma2_IOT file.

3 Active Window Requirements

Let's review our requirements for Active Windows:

- PCB's and motors reside within standardized modules that support replacement. In many cases, they are accessed after opening a hatch or removing front perimeter wood molding secured with bolts. Standardized Modules and Accessibility enables one to maintain building over a long period of time.
- Support multiple motors (e.g. window up/down, rolled blinds, venetian blinds, curtains, and thermal covers)
- Support multiple indoor and multiple outdoor sensors.
- Connect to building network and building power
- Support rolled or solid thermal cover embedded in wall.
- Provide user interface.
- Provide five nines reliability (operational 99.999% of the time).
- Processor sleeps when not in use, to reduce power consumption.
- Support multiple markets, including residential, commercial and industrial.

3.1 Fitting It All Together Mechanically

How might one fit this together mechanically?

Let's begin by reviewing the various components that surround a window.

Internal wall framing typically consist of 2x4 (1.5 x 3.5") or 2x6 (1.5 x 5.5") lumber. Walls with more load (e.g. 3 story building), walls with internal pipes, walls with more insulation, and colder climates favor greater wall thicknesses.

The Window Casing, shown to the right in natural wood, is a box that sits between the room and the actual window (window is white in photo). In some cases, the window itself exactly fits the wall thickness; otherwise, one adds casing to make up the difference.

Drywall, on the internal room surface, if often between 0.375" and 0.625" thick.

As one can see, the thickness of a wall varies.

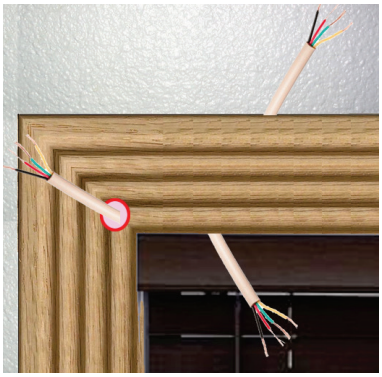
Window trim, shown to the right in white, wraps around the window and is visible to the room occupant. Typical width is 3", yet varies.

The window itself is often sold separately from the elements described above. The architect insures the various pieces fit together and are aesthetically pleasing.

Where does one physically place Active Window Modules, and how does one access them?

There are several options, illustrated below. Internal framing (e.g. 2x4 or 2x6) is shown in brown and module equipment bays at top, bottom, left and right are shown in red. Also, a box for rolled thermal covers, embedded in the wall, is shown in blue.

Equipment bays are covered by front perimeter molding; therefore, one can easily access their contents by removing molding secured with bolts. Framing is pushed outward due to additional width of bays. This illustration shows 4 bays, yet actual window products would probably have one or two.



Motors for curtains, rolled blinds, and venetian blinds reside somewhere near the window top. They could potentially be external from the equipment bay and built into a curtain rod, rolled blind shaft, or venetian blind upper mechanism. In these cases, a cable routes from equipment bay to external module via small port in molding top surface, molding bottom surface, or molding front surface; as illustrated above.

Researchers consider alternatives to equipment bays, such as cavities milled into wood windows and accessed via a hatch. For more ideas, see Chapter 16 "Create Standards that Automate Windows and Doors" in Ma2 Blueprint.

Researchers do not need to suggest standards for the actual windows and bays, since these products can vary and the building will still survive over long periods of time w/o losing value. It is the modules and their sockets that require electrical, protocol and mechanical standardization.

In order to insure a proposed standard works well in a system, researchers must: demonstrate hidden modules fitting into a window, is low cost, and supports replacement via removable hatches/molding. For this reason, researchers *do* need to think about equipment bays and how everything fits together mechanically and electrically, and demonstrate at least one complete system that works well.

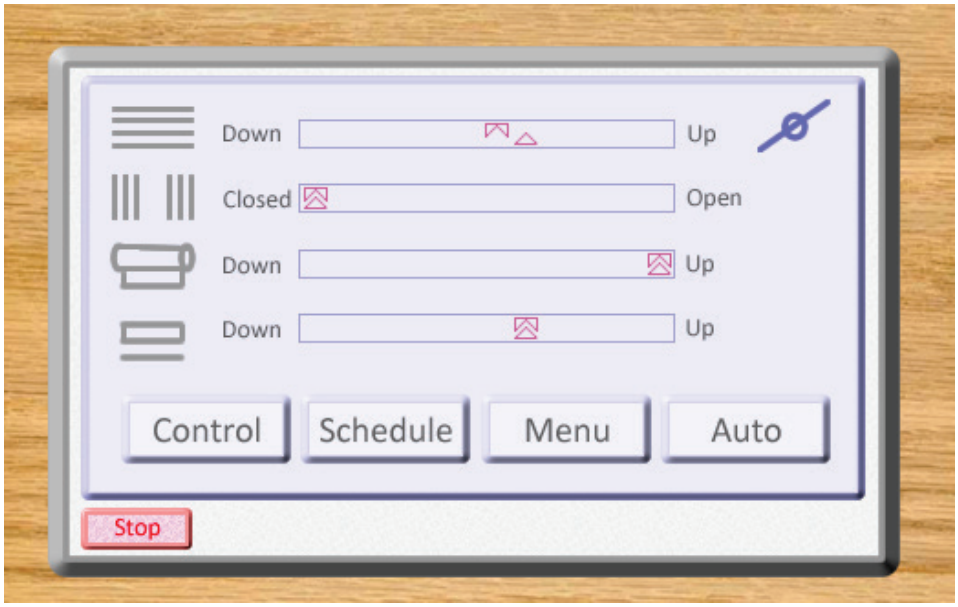
4 Different Types of Window Modules

Given our requirements, we can assume we have several categories of modules:

- Window Motor Modules (WMM) contain a standardized connector, PCB, motor, gears and sensors related to motor. This module would reside in an equipment bay, rolled thermal cover box (above window in wall), curtain rod, rolled blind shaft, or venetian blind mechanism.
- Window Control Module (WCM) includes user interface hardware (e.g. LCD touch screen), indoor sensors, outdoor sensors, networking interface to building, and interface to other modules (e.g. to motor modules).
- Window Power Supply Module (WPSM) converts building power (e.g. 48VDC, 110/220VAC) to voltages used by window modules.

5 User Interface

Researchers work on user interface, an example of which is shown here.



This presents a touch-screen LCD panel, and a mechanical "Stop" button that halts all motors (e.g. send CANbus command and turn off 24V power). This could potentially reside in a Window Control Module which mounts under, or to the side of, a window. Installation personnel would need to cut a hole in the molding (e.g. ~3" wide) to support this module socket.

In this example; venetian blind, curtain, rolled thermal cover, and window open/close control is provided in the LCD upper region. The set point symbol (△) indicates the desired position and the current position symbol (▽) reflects the actual position. The Venetian blind angle adjustment symbol ( ) adjust Venetian angle. The symbols along the left edge update to reflect current position. The Control button invokes the control screen, shown here. And the Schedule button presents a UI for establishing schedule. The symbols shown in this example might be confusing to an inexperience end user. What might one do to improve?

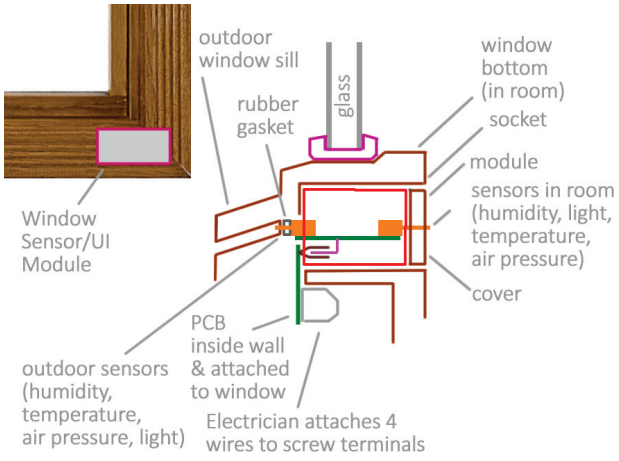
NEST is an example of an outstand User Interface; subsequently, researchers are encouraged to strive for this level of quality.

6 Window Control Module

The Window Control Module (WCM) includes user interface hardware (e.g. LCD touch screen), indoor sensors, outdoor sensors, network interface to building (e.g. BuildingBus 48V or AC), and interface to other modules (e.g. to motor modules).

The illustration to the right shows one way this module might fit into a window.

This module requires access to the outdoors, which might be implemented with a small physical port under the window, as shown in the illustration.



7 Window Power Supply Modules

The Window Power Supply Module (WPSM) in many cases (yet not all) would be separate from the Window Control module, since power needs can vary 10 to 1 depending on how many motors one has and their sizes. There are two types of WPSM modules. Those that attach to BuildingBus AC are powered by 110/220VAC, whereas those that attach to BuildingBus 48V are powered by 48VDC. Researchers evaluate the various options along with their costs, advantages, and disadvantages.

Power supply output needs to be standardized since other modules need to know what to expect. One option is the Power Supply module outputs 5.25V/100mA/500mW for $1 microprocessors within all modules (e.g. 1 to 4mA while sleeping and 4 to 16mA while operating, for each processor); and also outputs something like ~24VDC for motors. The 24V power would only be turned on when needed, to reduce quiescent power loss (typical power supply burns power when output current is low). Power supply modules might be available in small, medium, large, and extra-large (e.g. 250mA/6W, 500mA/12W, 1A/24W, 2A/50W; or perhaps larger). The Window Control Module might sequence motor operation if power capability is exceeded (e.g. only drive one motor at a time).

If 5V is sent to microprocessors and they need 3.3V, one might convert with a switching regulator for $0.35 in parts (e.g. #AP3429AK) or low drop out voltage regulator for $0.10 in parts (e.g. #TLV74133).

It is much easier to drive MOSFETs with 5V than 3.3V; however a 3.3V microprocessor can still control 5V power with the help of a buffer that is powered with 5V (e.g. #SN74LVC14APWR, $0.09).

If you run 2A through 22 gauge wire (16mΩ/ft), for 20ft, for example, then you will get a 0.6V drop on your Gnd wire and a 0.6V drop on your 24V Power wire. This voltage drop on the power wire is manageable. Yet the voltage drop on the Gnd wire effects your 5V power and CANbus transmission. One might be ok with 0.6V, yet not too much more. Researchers consider different wire sizes, advantages, and disadvantages.

8 Window Motor Modules

The easiest way to drive a DC motor is to apply a DC voltage to a coil via transistors or Mosfets. Also, one can reverse +/- polarity to reverse the direction. For details, search for "motor control" in file Ma2_IOT.

AC motors typically have 3 inputs: COM, Forward and Reverse. One attaches VAC neutral to COM and VAC Hot to Forward_Input to drive in the forward direction and VAC Hot to Reverse_Input to drive in the reverse direction.

There are a variety of techniques to determine motor position. A simple method is to sense current and run motor until a current spike, indicating end of travel. Researchers explore the various options, costs, advantages, and disadvantages.

Motor modules might contain a $1 microprocessor that supports CANbus and provides multiple analog input channels. For a list of M0-series processors with CANbus, click here. The lowest cost M0-series microprocessor with 16KB of ram is the #XMC1403Q040X0064AAXUMA1 at $1.27, for example. For details, search for "STM32 Processors" and "Infineon Processors" in file Ma2_IOT.

Sensors within motor modules might include things like: voltage measurement of 24V power, current flow through motor, torque measurement, and temperature. Control might involve things like driving a transistor or MOSFET that sends power to a motor. Commands might be something like: move from current position to X % open within Y seconds.

Lifting a heavy window quickly requires significant power, especially if stuck. Obviously, gears can increase torque. Moving lighter elements such as a rolled blind might require 1/20th as much power.

One needs to take into consideration safety, especially when closing windows. Especially heavy windows where it might be difficult to discern between a sticky window and a person. Researchers consider different ways of improving safety when closing a heavy window. For example, one might mechanically adjust the seal between window edge and frame, and rely more on gravity than motor force. Or place sensor(s) at window bottom edge and look for fingers.

The automotive industry has much experience with moving windows via motors/gears. For details, search google for: window motor module. Also, one can search google for: power window control. As one can see from the auto industry, these involve significant forces and need to be secured with bolts.

The $1.27 Xmc1403 microprocessor, for example, is capable of stepping motors and also doing DC-to-DC conversion with voltage limiting, power limiting, and current limiting. Researchers consider the various things one can do w/ these processors, paying attention to costs and benefits.

Researchers also consider the various options for motor control, and study multiple options, including variable speed and variable torque.

9 Physical Window Thermal Covers

Rolled thermal covers are thin and flimsy, and are therefore likely to degrade over time (e.g. cat scratches or tears surface); subsequently, standardization is needed to get the building through 100

years. Solid thermal boards (e.g. 1" to 2" thick foam), on the other hand, are more rugged, and therefore probably do not require standardization.

Solid foam insulation typically provides R6 thermal conductivity per inch of thickness. Consequently, 2 inches of thickness provides R12 total, for example (6x2). The average wall in America is R16, therefore R12 is close to wall. Alternatively, if a rolled thermal cover has R5-per-inch and is only 0.3" thick (due to little space in wall above window), then total works out R1.5 (5 x 0.3), for example. In this example comparison, solid foam thermal conductivity is 8-folder better than rolled. Rolled covers can improve if they also seal airflow (older windows often leak) and if they provide a reflective surface that reflects heat radiation. Researchers consider different ways of reducing energy loss at active windows.

Air conditioners have standard sized filters (e.g. 24x24x1", 20x20x1") and are easily replaced. Researchers look at doing something similar with rolled thermal products, with standard widths and thicknesses. It is not clear if lengths need to be standardized as well, since one might see multiple rolls at hardware store (e.g. 32", 36", 44" widths) that are cut to length by salesman. One might affix rails at top and bottom to help attach to shaft and mate with bottom window-sill groove. If one standardizes length *and* width, and window is shorter than cover, then excess material will remain in wall when fully deployed. In a sense, this is ok, yet this also limits the thickness of the material. Notice that 3.5" thick walls do not have much space, and therefore require thin material.

Stopping airflow and providing high reflectivity is an important part of thermal insulation. Researchers focus on thermal issues and not aesthetics when designing thermal covers. Researchers assume the public is not accustomed to thermal covers and will complain about how they look. Our strategy is to utilize an occupancy sensor to only deploy when room is unoccupied. And push public to accept poor aesthetics when unoccupied. Many Americans want their living room to look good even when they are not at home, which is an indication of our vast wealth. Our strategy is to ignore this logic, and press forward.

10 Industry Analysis

Window manufacturer (e.g. Pella, Andersen) revenue is increased by active window standards due to: (1) increased average selling price from product that "does more", and (2) increased incentive to replace current windows with new (i.e. "upgrade from passive to active").

Processor and Semiconductor manufacturers (e.g. ST, TI, Infineon) revenue is increased since they can sell more IC's.

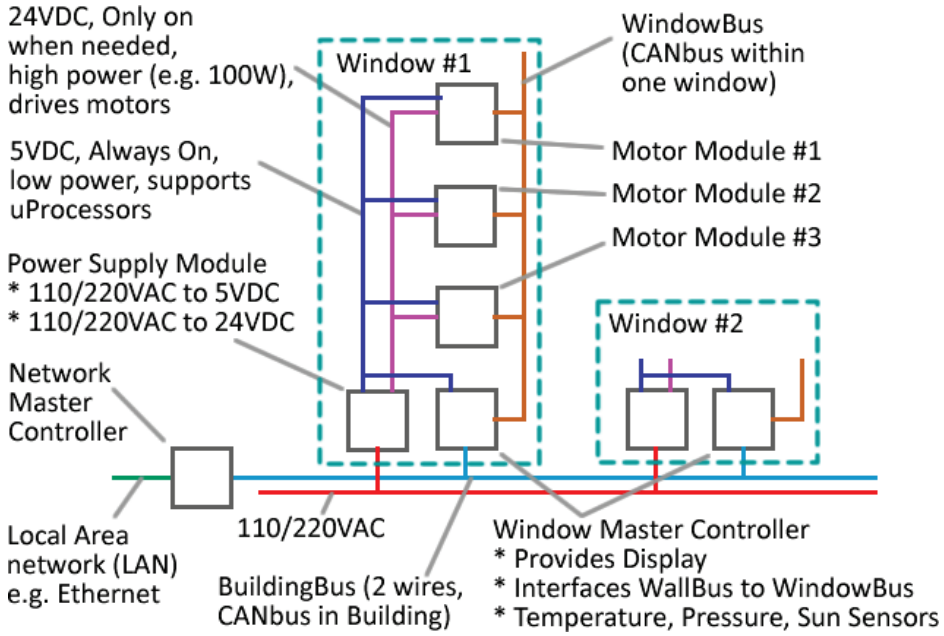
Governments benefit from active window standards since it helps them achieve new CO_2 emissions targets.

Architects benefit from active windows since it helps them meet new gov't requirements without reducing Window-to-Wall ratio.

Given all this demand, it is reasonable to expect researcher efforts to be utilized worldwide.

11 Fitting It All Together Electrically

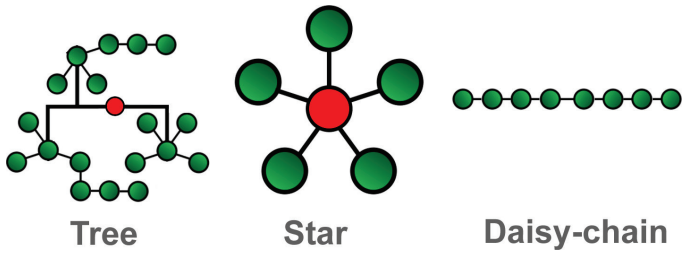
Window Control Modules talk to the building via BuildingBus 48V (48VDC) or BuildingBus AC (110/220VAC); and Modules within each window talk to each other via WindowBus. Both WindowBus and BuildingBus are based on CANbus, and are similar. The picture to the right shows 110/220VAC power routed to each window via BuildingBus AC. Alternatively, if one's power needs are less, they might consider BuildingBus 48V. Each window has a Window Master Controller that connects BuildingBus to WindowBus, provides sensors, and optionally provides a user interface display. Each Motor module is powered by 5V (for microprocessors, always on) and 24VDC (for motors, only on when needed).



12 BuildingBus Development Initiative (BuildingBus™, Building Bus™)

12.1 BuildingBus Overview

Manhattan 2 is developing a 2-wire communications bus that interconnects devices within a building, including things like fans, dampers, thermal window covers, and lights. There are two versions:



- BuildingBus AC (High Voltage, 110/220VAC power, ~2KW power/network)
    - Typically supports 2,000 Watt loads per network (e.g. 20A at 110VAC)
    - 5-wire cable includes: 110/220VAC neutral/hot, Earth Ground, ACBB Data+-
    - ACBB Data+- wires are used for communication and provide the following features: survives accidental connection to 220VAC (+-400VDC), one can lose earth ground connection between devices without degradation, ACBB Data+/- wires can be swapped without degradation, 110/220VAC neutral/hot wires can be swapped without degradation, and supports microcontroller CANbus communication. This probably requires opto-couplers to resist large common mode voltages between devices and potential loss of an earth ground connection.

- BuildingBus 48V (LV, Low Voltage, 48VDC power, ~200W power/network)

- Supports ~1/10th the loads supported by BuildingBus AC (e.g. 4A at 48VDC, 200W).
- Involves smaller wires and low voltage building codes (less conduit).
- 5-wire cable includes: 48VDC power+-, LVBB Data+-, Shield/Sense (no current, connected to earth gnd, shields RFI, common mode voltage sense for CANbus).
- LVBB Data+- wires are used for communication and provide the following features: survives accidental connection to +-48VDC, supports +-10V common mode difference between devices (opto-couplers not required), requires proper connection of ground wire, and supports microcontroller CANbus communication.

LVBB Data+- and ACBB Data+- both utilize CANbus and both use the same communications protocol (layer above CANbus). The only difference is the electrical signaling physical layer (e.g. what voltage constitutes "0" and "1). LVBB and ACBB can be mixed with the help of a tiny PCB that translates between the two.

BuildingBus AC focuses on large loads that includes fans, large motors that move heavy windows, and many or large LED lights.

BuildingBus 48V focuses on smaller loads that includes motors that move physical window curtains/blinds/thermal covers, motors that adjust duct/vent dampers, smaller or fewer LED lights, sensors, light switches, and user interface panels.

BuildingBus AC cables contain bulky 110/220VAC power wires (e.g. 10 to 16awg for hot/neutral/earth), whereas BuildingBus 48V cables are lighter, easier to handle, and less costly (e.g. 18 to 20awg for 48VDC power+-).

Both BuildingBus 48V and BuildingBus AC provide the following features:

- Module processor sleeps and draws little power when not in use, and wakes up after it receives CANbus frame.
- Wiring supports tree topology (daisy chain not required). This is similar to branches on a tree as opposed to one wire in a line with termination at both ends. With tree topology, there are no termination resistors (there would be too many). With no termination, one needs to reduce rise/fall times to avoid ringing.
- ≥ 99.999% reliable (not wireless, not power line communication).
- Transceivers consume ≤ ~10mW of power (e.g. 3V/3mA) when signaling (as opposed to ~10x more utilized by RS-485 and traditional CANbus).
- Any wire can touch any wire in cable without damage.
- System is hot-socket compatible (insert module into socket and any combination of wires can make contact w/o damage).
- Supports CANbus.
    - When two or more devices transmit at the same time they combine using wire-AND logic (i.e. we see "0" on bus if at least one device transmits "0"; otherwise "1"). RS-485, for example,

does not do this since the driver with the most current output dominates (i.e. logic level is not predicable when two transmitters are in contention).
    - CANbus requires that the rise time and fall time to each be < 25% of the bit duration (e.g. < 7uSec with 30K bps).
    - For details, search for "CANBus Physical Layer Signaling" in file Ma2_IOT.

Electrically, ACBB Data+- (BuildingBus AC) might look similar to DALI, where optical isolators electrically decouple data wires from Device COM. For details, search for "Propose DALI 3 electrical signaling standard" and "DALI Over-Current and Over-Voltage Protection" in file Ma2_IOT.

Electrically, LVBB Data+- wires (BuildingBus 48V) might look similar to traditional 120Ω CANbus, yet with slew rate control and no 120Ω termination resistors. For details, search for "Develop New Electrical Signaling Standard" in file Ma2_IOT; and see below notes on "WindowBus Physical Layer Signaling Standard".

Bit rates and maximum network length are To Be Determined (TBD).

For fun, let's go through the math with a 200ft maximum network size: 200ft x 2nSec/ft = 400nSec flight time, 10:1 rise-to-flight ratio avoids ringing (or 20:1), 10 ratio x 400nSec = 4uSec rise/fall time, 30Kbps has 30uSec bit duration and 25% of this is 7uSec (CANbus wants ≤25%), 4uSec rise time + etc. delays is ≤ 7uSec, 200ft x 20pf/ft = 4nF, and 1000 ohm source impedance driver x 4nF = 4uSec time constant (e.g. 1K resistor between Data+ and Data- to pull to 0V when not driven). Researchers might consider a driver with slew rate limiting, or a fast driver connected to a 1000 ohm series resistor.

Researchers propose standards that define physical layer (e.g. LVBB (48V) might be CANbus-like, and ACBB (110/220VAC) might be DALI-like), data layer (e.g. CANbus) and protocols above data layer (e.g. KNX, or custom protocol designed by researchers).

Is 48VDC a good voltage for power? Is a shield/sense wire needed with BuildingBus 48V? Are opto-couplers needed with BuildingBus AC? Is twisted pair needed with data wires? How much common mode voltage needs to be resisted between devices? Researchers consider the various options, along with their advantages / disadvantages.

12.2 Replace and Run via Sockets with Memory IC's

BuildingBus supports "Replace and Run". This means that one can Replace a component on the network and run away while system runs ok (no set up required). Each component (e.g. damper, fan, light, and motor in window) is attached in parallel and therefore the network master controller and the component module has no idea where they are physically located in the system.

BuildingBus components plug into physical sockets and sockets contain a tiny memory IC. Two wires connect the memory IC to the device (Memory+/-). The electrician who wires the building installs cable, attaches it to sockets, and then programs the sockets. Programming entails telling the socket where it is physically located (e.g. "living room, building front wall, vent"). It is much easier for the electrician to do this once for all sockets when the building is first constructed, than for others to do this for each device,

each time a device is plugged in, over a building's lifetime. The electrician is familiar with the system, whereas people that service the building over its lifetime (e.g. homeowner) might not be adept at performing setup.

We envision BuildingBus being used with many devices within a building, as opposed to a homeowner do-it-yourselfer who visits Home Depot, buys two devices, and installs themselves. Subsequently, it makes sense to push setup toward the electrician who wires the building, connects the wire to the sockets, and is familiar with the system.

WindowBus Modules (modules inside an active window) plug into sockets as well. Consequently, active window manufacturers install sockets for modules, program the memory IC's in those sockets, and cable them together. Later, when modules are installed or replaced, they will know where they physically reside within the window. This is especially important with active windows that leave the factory with empty sockets cabled together, yet no installed modules ("active window *capable*").

Metaphorically, the system talks to "sockets", not devices, and devices reside at sockets.

The memory IC also helps to support Replace and Run by providing a place for modules to store state information. Below are example memory IC's:

- AT21CS01-UUM0B-T, 2wire, 1x1x0.5mm, 128 bytes, $0.22
- 11LC040T-I-TT, 3wire, 4kBytes, $0.18.
- See also: 1Wire Interface, Maxim 1Wire Products, and I2C 1Wire Memory.

13 Our Unique Requirements

We have three unique requirements that have enormous impact on our system design: Fault Tolerance, No Engineer, and Slow.

13.1 Fault Tolerance

Fault tolerance means that if something breaks, little disruption occurs to the system. To do this, we want for decision making to occur as close to the target device as possible. For example, if an occupancy sensor sees someone frying an egg below, and a ceiling light is 1meter away, we would like for that light to turn on, even if there is network disruption elsewhere. Another technique that facilitates fault tolerance is we break up the system into sub-networks and multiple networks. Subsequently, if one goes down, others can continue. This entails devices being routers to move messages through the system. For example, one might send a message to a Window Controller device, which in turn sends it to a Motor Module device within that window. If the sub-network within the window becomes faulty, we would like that fault to be contained within the window.

13.2 No Engineer

No Engineer means that the general contractor that constructs a building is not expected to employ an engineer to design the automated system. Alternatively, traditional networking protocols are typically set up inside machines (e.g. CANbus inside a car) or on a factory floor (e.g. Tesla factory building cars), and they all entail engineers that design the system. For example, CANopen software works with a table of registers and .ini configuration files that include addresses of internal hardware; and the engineer works with addresses and registers to design the system. If you are at Toyota Motors and designing a car, this is all ok. However, in our case, we want massive adoption of low carbon living worldwide, which means we want low cost of entry, which means No Engineer.

13.3 Slow

We are physically running data wires alongside power wires, which means our network is a tree topology, instead of the traditional daisy-chain or point-to-point (e.g. Ethernet). This means we need to slow down signal rise and fall times to stop ringing, which means we need to run at ~100K bps instead of the more traditional ≥ 1M bps. To make this work, devices utilize the network sparingly, which means they need to be somewhat intelligent. An example is occupancy sensors and light sockets all gather information about devices in the room, all maintain status information on each, and all participate in decision making. When a person exits the room, the occupancy sensor broadcasts the change, and devices update accordingly, all via one 8byte message packet.

13.4 BuildingBus Framework

There is only one way to satisfy our unique requirements, and that is to supply source code that is placed onto the devices. Other networking protocols do not do this, and this is perhaps the largest difference between BuildingBus and other systems. BuildingBus is also Software, and not just a networking protocol.

The BuildingBus software is referred to as "The BuildingBus Framework". It manages devices, routes packets, gathers information about nearby devices, transfers device information to higher level devices, reports sensor measurements, and supports local decision making.

The engineer who designs a BuildingBus device (e.g. Dell Corporation programmer developing BuildingBus Motor Module product) does not need to understand the BuildingBus framework, and instead can focus on setting fields that characterize their product, and filling in the bodies of functions that do things (e.g. move motor, adjust illumination of light bulb).

Companies look at their revenue and think about how to increase. We, on the other hand, look at worldwide CO_2 emissions, and think about how to reduce. This difference causes us to approach design problems differently. We think about how to coordinate others, whereas companies think about how to control others, to increase revenue. Our unique position enables us to do things that companies cannot. A company cannot propose that their propriety system be the basis for automating buildings since companies, governments, and customers will resist their control. Alternatively, university researchers

can do exactly that via free and open technology. In conclusion, university researchers are in a unique position to significantly coordinate infrastructure within futuristic buildings, to reduce CO_2 emissions.

14 WindowBus Development Initiative (WindowBus™, Window Bus™)

Researchers develop WindowBus™, which is a standard that defines how modules talk to each other within one window. This is similar to BuildingBus, described previously, yet involves shorter distances. WindowBus connects modules in parallel via multiple ~22awg wires, possibly in a ribbon cable. Below is a list of wires that one might place in this cable:

- 5.25V power: ~100mA total to all modules, always on, used by microprocessors, 5.25V+-0.25V at power supply, 4.75V+-0.25V at module when motors are on, regulated, isolated from 110/220VAC.
- 24V power: 250/500/1000mA total to all modules, unregulated 20…25V, isolated from 110/220VAC, only turned on when needed.
- Power- wire (returns current from power).
- Shield/Sense Data (hopefully no current, shields RFI, common mode voltage sense for CANbus, connected to BuildingBus AC EarthGnd or BuildingBus 48V Shield/Sense).
- One CANbus data wire.
- One wire connects module to memory IC embedded in socket.

WindowBus is similar to BuildingBus in the following ways: CANbus data layer, devices use little power when not in use, wiring supports tree topology, ≥ 99.999% reliable, transceivers consume ≤ ~10mW when signaling, any wire can touch any wire in cable without damage, system is hot-socket compatible.

14.1 Researchers Develop WindowBus Physical Layer Signaling Standard

Small $1 microprocessors have two ways to communicate with a network: serial UART and CANbus. The CANbus controller handles framing and media access control; and is therefore much more useful than a UART. Since it is free (i.e. already built into the processor), it makes sense to use it.

Devices that talk to each other on 2 wires typically use RS-485 or a 120Ω CANbus electrical signaling physical layer. However, these have several issues: damage occurs if short to 110VAC, does not support tree topology wiring, and requires ~50mA to drive whereas we would prefer 1/10th as much.

Researchers consider alternatives such as: terminate at one position with one 1KΩ resistor (between Gnd and Data wire), drive with 3mA, set driver rise/fall time to 2.5µSec (e.g. op amp with feedback?), transfer data at 100K bps, and support 0 to 7meter network sizes (i.e. one physical window). Implementing this form of CANbus might involve an op amp (e.g. #LMV321A, $0.10) and SPST switch (e.g. #NLAST4501D, $0.12, connects op amp to bus) for Transmit, and a Schmidt Trigger (e.g. #SN74LVC1G17DBVR, $0.03) for Receive. 2.5uSec rise time will not ring if flight time is 1/20th as much (125nSec = 2.5uSec/20, 120nSec / 2ns/ft = 62ft maximum network size). Total cost is $0.25/2mA, which is decent.

Alternatively, one might look at a transistor or MOSFET that lifts the Data wire to 5V power via a 1KΩ series resistor. If one combines a 74LVC14 ($0.10, 6x Schmidt trigger inverters) for receive and mosfet gate drive, and a $0.03 MOSFET, then total parts cost would be $0.13. For an example of this circuit, see this pdf schematic and TINA simulation file. Notice simulation shows no ringing with 1KΩ series resistance, yet ringing with less resistance.

For an example of a one-wire signaling system, see this pdf schematic and TINA simulation file.

If cable is 30pF/ft and 30ft long, then capacitance between Data and Gnd would be 900pF (30*30). If line is not driven actively and instead uses 1KΩ termination resistor to move to the logic "1" position, then fall time would be 1µSec (1KΩ x 900pF), which is reasonable with 100K bps signaling.

If one wanted to resist inductively coupled spikes (e.g. current turning on in one wire couples sharp spikes into neighbors), they might consider a low pass filter at the receive circuit. For example, a 166 KHz 1-pole low pass filter would help resist ≤300nSec spikes given 30Kbps signaling (1µSec time constant, 166 KHz = 1 / (6.28*1µS)). One could look at doing this with a capacitor after passing through an input series resistor (i.e. capacitor is not tied directly to data wire since multiple capacitors would load excessively). To protect against 100V overvoltage, one might consider a 50KΩ series resistor (0.2W = 100^2/50e3, 2mA).

Several op amp products offer shutdown capability, yet turn-on time tends to be longer than what we would like, and we need output to enter high impedance when shutdown. The MCP2551 CANbus transceiver, for example, supports slew rate control, yet is a bit more than we need (i.e. 75mA, $0.82). The op amp supports feedback, which means it can keep the rise/fall time at 1µSec, even with greatly varying capacitive loads.

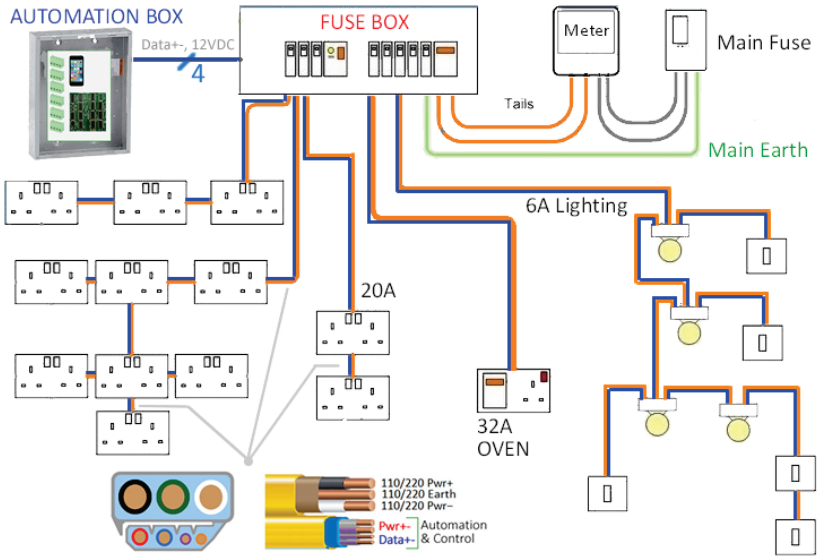
Is 5V a good choice for always-on power? Is 24VDC a good choice for switched motor power? Would it be better to have one power supply voltage instead of two (note that quiescent power on larger power supplies w/ low output current might be an issue)? Should we consider supporting the cabling of 110/220VAC to motors? Do we need a separate wire for 24V return current that sees voltage drops when motor is on? Researchers consider the various options, along with their advantages / disadvantages.

Researchers consider these techniques; and others; in search of a low cost, reliable, rugged, and low power technique for connecting multiple $1 microprocessors within a window. For details, search for "CANBus Physical Layer Signaling" in file Ma2_IOT.

15 Smart Building System Design

15.1 In the future, your house will be more like your car

We are looking at connecting windows to buildings, yet this begs the question, "What is the over-all system design and strategy for the entire building?" We are looking at utilizing CANbus to network together devices within buildings, since this is built into many low cost microcontroller IC's. CANbus is the de facto standard used by the automobile industry, so we are looking at turning your house into your car, in a sense.

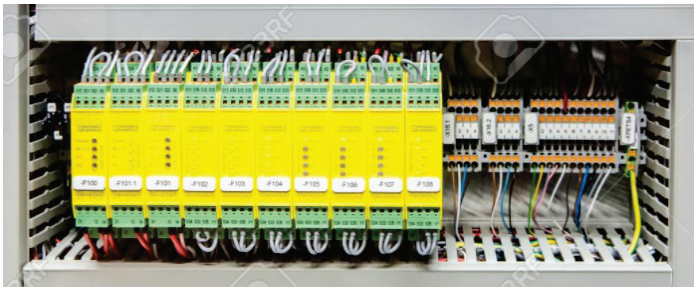


15.2 Installing BuildingBus cable cost little

Shown here is the power wiring for the typical house. 3-wire power cables emanate from a master fuse box (e.g. in house basement), where each cable is associated with one fuse. Cables route to electrical boxes in a tree topology.

It is likely that builders will add 2 wires to this 3-wire power cable, illustrated to the right. This additional wire is not expensive -- 1000ft of 2-wire 18awg cable sells for $140 at Amazon. The labor cost of installing the cable has already been paid for; therefore, installing the 2-wire data network to the point of the electrical boxes is low.

15.3 Master Automation Box



Notice in the above illustration, the power wires terminate at a fuse box, possibly in the basement. If you add 2 data wires to the 3-wire power cable, you would end up terminating your data wires at this location, since that is where they are. One might place an "automation box" next to the fuse box, pictured here, where plug-in modules manage each BuildingBus network, one module per network. The modules (i.e. "BuildingBus Master Controllers") would then be networked together (perhaps by CANbus) within the card cage, and this then might attach to the local area network via ethernet (IP). In summary, each 3-wire cable (augmented to 5) is associated with one 110/220VAC fuse, one 2-wire BuildingBus network, and one BuildingBus Master Controller module.

15.4 BuildingBus Sub-Networks

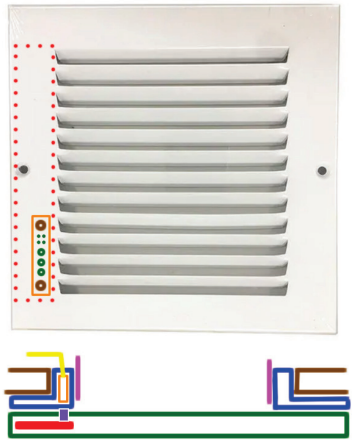
BuildingBus is a 2-wire network that supports adding a device that is a gateway to another network, called a "sub-network", which is its own CANbus network. The 2 CANbus wires in the sub-network do not touch the main network data wires. Instead, one sends a message to the gateway device, who repeats it on the sub-network. The reason we don't put sub-network devices on a BuildingBus network is two-fold: (1) reduces network traffic by reducing # of devices on one CANbus network, (2) reduces risk of one faulty device effecting others. The Xmc4200 processor, for example, has 2 CANbus controllers, and can therefore implement a bridge between two networks.

22.12.1WindowBus™ -- Thermally Convert Your Window to a Wall

As noted previously, WindowBus is a CANbus network that connects together multiple modules within a physical wall window. The gateway device for WindowBus is the Window Master Controller (WMC) module, which connects a BuildingBus network to a local WindowBus sub-network (one per physical wall window).

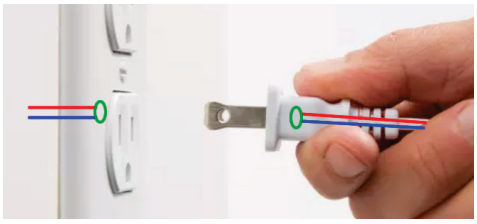
22.12.2VentBus™ - Regulate Temperature at Occupant Position

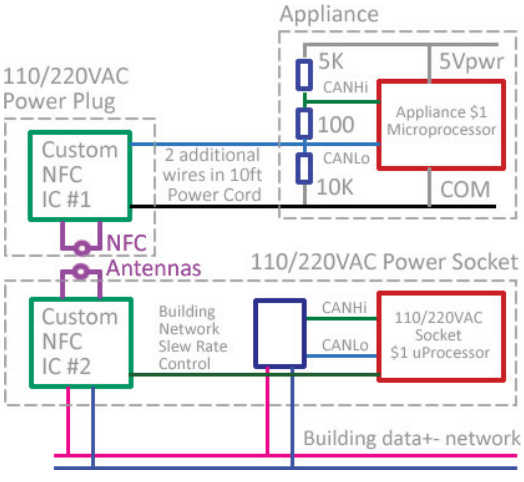
An HVAC vent could be augmented with a motorized damper via a BB device PCB that resides under the vent grille. This PCB might have sockets for sub-network modules that do things like: occupancy sense, fire detection, security camera. This would enable one to control the temperature of each room (multiple zone). And with the help of an in-line duct fan connected to BB, one could move air from one room to any other room, without turning on the central HVAC system. Perhaps "VentBus™" modules are physically standardized in a manner similar to that done with windows? For details, see Ma2 Fan/Damper initiative.



22.12.3PowerPlugBus™ - Your Outlet and his Friends have a Lot to Talk About

One might also place a *sub*-network at an electrical box. Each power socket might contain a current sensor that allows one to monitor the power at each outlet. This would enable one to understand which plug-in hardware consumes significant energy, which might encourage less use.



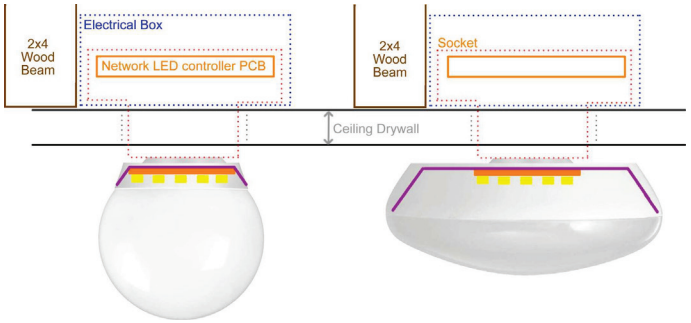


Also, it is our intent to develop and propose a standardized and reliable method that connects 110/220VAC hardware (e.g. vacuum cleaner, washing machine, refrigerator, etc.) to the data network. If one wants to coordinate appliances (e.g. connect refrigerator to 58°F ground source water), you need to be able to talk to them with 99.999% reliability (not Wi-Fi). For details on how this might work, see

illustrations here, and search "Develop Near-Field Communication between Socket and Corded Device" in file Ma2_IOT.

22.12.4IlluminationBus™ -- Illuminate Only As Needed

If one has multiple LED bulbs in their living room ceiling and wants to control each with 0 to 100% illumination, they might considering connecting sockets together with a sub-network, and interface the sub-network to a BuildingBus network via a gateway device. Then, one could illuminate as needed (only), with the help of occupancy sensors.



Currently, blubs contain AC-to-DC electronics in their base, they run hot, and they therefore burn out easily. We envision a different system: the bulb is LED-only, the socket does power conversion (not bulb), sockets are powered by 48VDC (or 110/220VAC, yet 48V is easier), ceiling cables are lighter (saves money on parts and labor), and building codes are less stringent due to lower voltage (saves money on labor). If one drives an LED with 48VDC instead of 110/220VAC, they can avoid electrolytic capacitors (burn out easily), achieve better conversion efficiencies, run less hot, and achieve higher electronics longevity. With a new type of LED-only socket, mechanical engineers can add features that move heat from LED semiconductor to socket, to improve LED longevity, as illustrated here (notice how LED thermally connect to socket). Making this work would require a method of communicating LED current rating to socket (e.g. 10W, 3.3V, 3A), which we intend to develop (e.g. eeprom memory IC in bulb talks to socket). For lighting ideas, see article parts 1, 2, and 3.

It is our intent to develop and propose electrical and mechanical standards that govern a new system for lighting.

16 Existing Networking Standards

Researchers look at connecting Active Windows to existing Networking standards.

KNX is an existing protocol that typically uses RS-485 and token passing to talk to devices. Researchers consider demonstrating an Active Window as a standard KNX/485/token device. This has issues, as mentioned previously. Researchers consider modifying this to KNX/486P/CANbus.

DALI 2 is an existing protocol that typically uses the DALI 16V/250mA scheme to talk to devices. Researchers consider demonstrating an Active Window as a standard DALI 2 device. This has issues, as mentioned previously. Researchers consider modifying this to DALI 3P/CANbus.

Researchers consider demonstrating Active Windows with other networking protocols, such as BACnet and Zigbee (connecting window to network via wire).

Somfy is an example of an existing propriety system that utilizes KNX/485/token to control rolled blinds. It is expensive, it does not offer thermal covers, and it is not a free and open standard accepted worldwide; yet it is probably the closest to our proposed system.

17 Active Window Development Strategy

Initially, researchers clip together existing Arduino and Mikroe Click boards, making use of the following existing products:

- DALI-2 Click Boards
- RS-485 Click Boards
- Temperature and humidity, weather, environment, and pressure Measurement Click Boards
- CANbus Click Boards
- Motor Control Click Boards

Adaptors connect Click Boards to microprocessors boards, as shown here.

For details on microprocessor boards, search for "Infineon Arduino" and "STM32 Arduino" in this file. Reasonable choices are the ST #NUCLEO-F401RE with #STM32F401RET6 processor, and the Infineon #KITXMCPLT2GOXMC4200TO with #xmc4200-f64k256-ba processor. These include plenty of RAM, Flash, A/D channels, and floating point hardware. Researchers initially work with somewhat powerful processors (e.g. $2 to $3 instead of $1) since they provide better support for monitoring, logging, and debugging. Design-for-manufacture and consolidation is a later step.

Researchers that develop interface hardware (i.e. protected-DALI 2, protected RS-486, RS-485P and DALI 3P) build prototype PCB's that fit the Mikroe Click form factor. This enables them to plug into existing prototype setups (e.g. replace existing RS-486 Click Board with protected version). Researchers that develop power supply hardware utilize the Arduino form factor. This enables them to place on top of, or next to, Arduino processor boards.

During years #1 and #2, researchers develop active window hardware and software technology. During years #2 and #3, researchers (possibly different teams) consolidate prototypes onto dedicated PCB's, polish and document software, write proposed standards specifications, and make available all work to others (e.g. offer Click designs to Mikroe Company free of charge to encourage adoption).

18 Development Requirements

Researchers work within the following framework:

- Device communication is "five-nine's" reliable (e.g. operational ≥99.999% of the time). This means no wireless, no power line communication, no batteries, and no solar panel powered electronics. Faults are expensive to resolve, especially when users are not familiar with

technology. Wireless is one to two nine's reliable due to obstruction, multi-path and crowded spectrum. Solar panels sometimes do not receive enough light. Power-line communication must contend with voltage drops along power wires and messy paths through fuse box. RS-485, DALI, and CANbus are all considered five-nines reliable.

- Processor sleeps and draws little power when not in use.
- All software is written in C and C++ (not Python) since this is considered the standard in industry. C/C++ compilers produce compact (unused fragments are eliminated) and fast code for small processors with limited RAM and FLASH.
- Electrical simulations are performed with TINA software since it is free and therefore easy for industry personnel to benefit. For example, research develops power supply that supports 3.3V/16mA, yet industry engineer needs 5V/25mA. With TINA, industry engineer downloads software for free from TI and adapts researcher's .TSC simulation file.
- Researchers maintain one spreadsheet with notes and calculations that is shared w/ others.
- All developed source code, schematics, simulations, spreadsheets and IP is made available free and open to encourage collaboration and adoption of standards, to reduce worldwide energy consumption.
- All developed technology is offered to multiple companies via email, free of charge, to encourage adoption.

19 10 Year Plan

Within the next few years, no one can make money on this technology even if they invest heavily since it involves many different parts; and one cannot develop those in short time. Since payback time is long, companies currently will not invest.

The only way to move forward is for university researchers to develop a rough mechanical and electrical design, and give it away for free to encourage utilization. Researchers can do mechanical simulations, perform electrical simulations, develop cost models, build simple mechanical prototypes, develop PCB's, and propose standards that facilitate plug-and-play standardization across multiple industries and companies.

We envision several phases. Phase I: researchers create prototypes and propose interconnection standards. Phase II: standards body formalizes standards (e.g. year's #4 and #5). Phase III: companies make use of standards (e.g. manufacture modules, years #6 and #7). Phase IV: world buys and installs. The world needs I before it can do II, needs II before III, and needs III before IV. We aim to do Phase I. This is our 10 year plan.

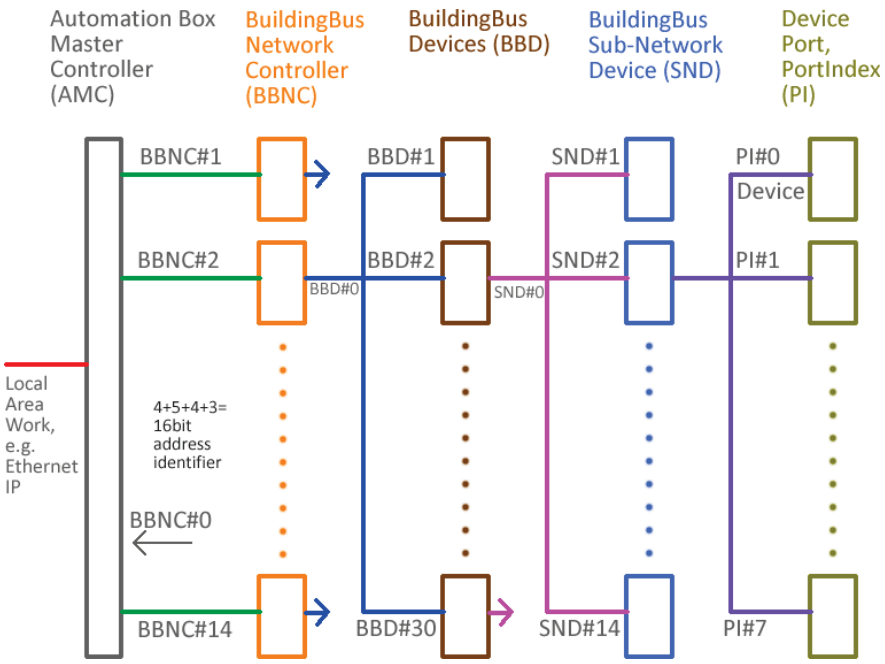
20 R&D Teams

Research is conducted by multiple teams, each of which assumes a different area of responsibility, as described in Active Window Teams, Fan/Damper Teams, and Solar BiPV/LiPV Teams.

21 Network Strategy

21.1 Network Topology

Previously, we talked about wiring-topology, which is how physical wires are routed. Also, there is something called "network topology", which describes how elements interconnect within a network. Our decisions are driven by: existing 110/220VAC wiring, needs at specific locations (e.g. windows, vents, power outlets), and CANbus controllers built into low cost processors. If we put this all together, we end up with a network topology that looks something like what we have here.



The Automation Box Master Controller (AMC) sits next to the fuse box and maintains 1 to ~14 BuildingBus Network Controller modules. Each of these modules drives a 2-wire BuildingBus network. Each network coordinates 1 to ~30 devices via CANbus. Each of these devices might support a sub-network (e.g. modules within a window, modules within a HVAC vent, modules connected to an electrical box), where 1 to ~14 modules attached to each sub-network (each sub-network has their own 2-wire CANbus "sub" network).

Also, each device has between 1 and ~7 ports, which are features within a device that one can interact with. When one sends a command to a device, or asks for information, they talk to a specific port (i.e. they specify a 3bit 0 to 7 PortIndex). PortIndex=0 refers to the over-all device, whereas PortIndex=1...7 refers to features within the device. Example features are: temperature sensor, pressure sensor, sun sensor, and motor.

For example, if PortIndex #3 is a temperature sensor, than a command to FunctionCallCmd (PortIndex = 3) returns the temperature if that sensor. If PortIndex #4 is a motorized physical window curtain, then a command to FireAndForgetCmd (PortIndex = 4) might tell the curtain to open a variable amount, as determined by a 0 to 255 value.

The illustration shown here refers to one "BuildingBus System". When one exits via IP (with faster and larger packets), they are leaving that one system. One can network together multiple BuildingBus Systems, via IP, yet that is beyond the scope of this document.

The numbers mentioned here are a starting point. Researchers can look at this and think about what makes the most sense. Keep in mind that too many things on one cable is risky. The cable might be physically cut at each device and or one might have a physical connector at each device. If one physical

connection fails, everything attached to the cable may fail. Also, one faulty device might interfere with all devices along the cable. Subsequently, we keep the maximum number of devices somewhat low and encourage users to not try to do too much with one 2-wire cable.

21.2 Messages Wiggle through System

In some cases, a message will move on the CANbus wires and hit their target on those wires; however, in other cases, the message wiggles between network levels, facilitated by controllers that read the message on one CANbus cable, and repeat it on another. The Xmc4200 does have two CANbus controllers, and is therefore capable of maintaining two networks. Many of the existing CANbus systems, and software, expect one CANbus cable. They are utilized with things like cars, trucks, and machines; which are physically smaller than a building.

Messages are repeated by controllers to move them to their ultimate destination, similar to a packet being routed in the internet, via router hardware. Here is an example of a BuildingBus wiggle. A message from a sub-network device controlled by device #1 on network #2 (BBNC=2, BBD=1) sends a message to a sub-network device controlled by device #3 on network #4 (BBNC=4, BBD=3). This would entail: Initiator sub-network device sends message to sub-network controller (BBNC=2, BBD=1), which repeats message to network controller (BBNC=2), which repeats message (again) to AMC (master controller, BBNC=0), which repeats message to network #4 controller (BBNC=4), which repeats message to sub-network controller (BBNC=4, BBD=3), which repeats it (again) to the sub-network device. Notice this involves 6 different transmissions, and is an example of a worst case scenario. In most cases, a message might be repeated one or two times.

21.3 Building Bus System

One Building Bus System is defined as one AMC that manages 1 to 14 BuildingBus networks and talks to the outside via IP.

21.4 Automation Master Controller (AMC)

Each system has one and only one Automation Master Controller (AMC). The AMC serves as a coordinator and interface to outside the system (via IP); however, much can be done without the AMC. For example, if one turns on a wall light switch then a command to a group of lights might only involve a portion of the system, without AMC interaction. If one has a small system with one BuildingBus Network Controller that manages one BB network and talks to the outside via IP (no automation box next to fuse box), then that controller takes on the role of the AMC.

21.5 BuildingBus Network Controller

The BuildingBus Network Controller might be a module in a physical Automation box next to the fuse box (e.g. in the basement), 0.3meters from the AMC processor. In this case, there are two primary ways it could interface with the AMC processor: (1) dedicated wires between AMC processor and each

controller (e.g. SPI to AMC FPGA or processor), or (2) all 14 controllers talk to AMC on the same high speed CANbus network (e.g. 8M bps, 12uSec per message).

If one wants to run CANbus between 2M and 15M bps they might consider CAN FD. The ATSAMV71Q19 is an example of a 2019-era $9 processor with 2 CAN FD channels, 1 Ethernet channel, 512KB flash and 256KB. This kind of processor might satisfy the requirements of an AMC that houses 14 BuildingBus Controller modules. Alternatively, one might look at a CAN FD interface IC (e.g. $1.30, TCAN4551).

It is also possible that the BuildingBus Controller resides far from the AMC processor (e.g. 100meters), and talks to the AMC via a high speed cable (e.g. 500K bps, 200uSec per message, two parties on one point-to point terminated cable).

There are advantages and disadvantages to the various approaches. Placing the controller in one location makes replacement easy and might match fuse box physical wiring (3-wire power cable with two BuildingBus wires ends up at fuse box). The advantage of placing the controller further away is the devices might be physically close together (e.g. 200 lights in one corner of large building) which entails a faster BuildingBus network (running tree topology long distance will slow it down; whereas point-to-point terminated non-tree with two parties runs fast).

The BuildingBus System is designed to supports *all* of the above options.

21.6 Devices

A device is a physical piece of hardware that attaches to the network, it contains a processor, and it talks to the system. In the illustration above, all rectangles are Devices. A 16bit address, called an "AddressDescriptor", is used to locate any device within the system. Each type of device is associated with a 32bit DeviceType code, which defines the type of device one is working with.

21.7 Ports

Internally, each device includes a set of "Ports" where they each reside at a "PortIndex" address between 0 and 7.

22.12.5DeviceCommon, PortIndex #0

PortIndex #0 contains a class called "DeviceCommon", which manages the entire device. This class is compiled into all devices on the network, which makes it easy for others to know how to interact with each device.

22.12.6DeviceTypeHandler, PortIndex #1

PortIndex #1 contains a class that relates to the DeviceType. If the module is a motor controller, for example, then it would compile this class and place it at PortIndex #1. Then, anyone on the outside would know how to interact with the motor controller. This is referred to as the "DeviceTypeHandler".

22.12.7Measurement System, PortIndex #2

PortIndex #2 contains the Measurement System class, which supports measurement of multiple sensors of any type.

PortIndex #3 through #7 are available to add any class one wants; however, the world might not recognize it, and therefore it might be ignored.

21.8 Data Capsules

Data is stored in a set of fields, within a C struct, called a "data capsule". This is portable, which means it can be moved around the network, and be understood by others. How this works is beyond the scope of this document. There are three types of capsules:

22.12.8Immutable Data

This never changes; therefore, you can copy this throughout the network once, and others can make use of it.

22.12.9Immutable Strings

This is similar to Immutable Data, yet consists of strings packed together. Nothing else. It provides an efficient way to move strings through the network.

22.12.10 Port Registers

This is similar to Immutable data (C struct with set of fields), yet the data can change.

21.9 Each Port has Three Capsules

Each port has one ImmutableData capsule, one ImmutableStrings capsule, and one PortRegisters capsule.

A FieldIndex enables one to access fields within a capsule. It ranges from 0 to number of fields minus 1.

Each field can be a single value, or an array of values. If an array, then one can access any element of the array by also specifying an arrayIndex.

Each field can be one of the following data types: int8, int16, int32, int64, uint8, uint16, uint32, uint64, flt32, flt64. Int8 can be used for ASCII characters. We favor signed integers over unsigned since some programming languages do not have good support for unsigned. For this reason, almost all fields are signed integer.

21.10Network Data Transfer Rates

Some areas of the network support faster transmission speeds than others. Tree topology wiring without termination (e.g. BuildingBus Network), for example, is forced to be slow to reduce ringing. Also, its maximum speed is determined by physical size (i.e. longest physical distance between any device to any other device on network).

Sub-networks tend to be smaller and can therefore run faster (e.g. 100K bps, 1mSec per message, ≤ 10meter physical network size) BuildingBus networks might run closer to 30K bps (e.g. 3mSec per message, ≤ 30meter physical network size). The BuildingBus Controller to AMC connection might run between 300K to 8M bps (12uSec to 300uSec per message). If one is working with IP (e.g. Ethernet), they might be looking at 3mSec to 300mSec per message, yet this would support large data packets and fast data transfer rates.

When doing real-time automation (e.g. measure one temperature, set vent damper to 50% open), a 30K bps 3mSec-per-message network might be preferred over 30M bps high speed Ethernet (1,000x faster) with 10mSec-per-message (3x slower).

An example of a different networking scheme is ZigBee, which communicates via Bluetooth wireless. Their packet sizes are 10 to 100 larger, and their data transfer rate (bits-per-second) is 10 to 100 times faster. They have their challenges. For example, 1% to 10% of the time (or more), their communication fails due to blocked signal, out of range, noise mixed with signal, or crowded spectrum.

21.11Less Talk, More Action

BuildingBus devices try to only talk when necessary, and try to do as much as possible with one message. BuildingBus is designed to do simple things reliably. For example, a wall light switch sends one message to a group of lights. Or a refrigerator sends one message to a pump in the basement requesting 4 liters of 58°F water. These are both one message operations. Less talk, more action.

21.12Errors

There are different kinds of errors. In one case, a bit error is detected during a transmission and a CANbus error frame from Target tells Initiator the message did not get through. If the message is wiggling through the network and this is one leg in a journey (i.e. Initiator is a device that is repeating the message), then the Initiator would need to send an error message to the originator (i.e. the CANbus Error frame only effects one leg of journey). Another type of error is the Target sees a problem within its software, and returns an error code to the Initiator. The ErrorPolicy parameter, described below, specifies how errors are handled.

21.13Manufacturer Server

Device might provide 32byte internet URL of manufacturer server in DeviceCommon ImmutableData capsule. This is a URL to a server, not a website. The server would need to conform to a standard proposed by researchers. The server could help coordinate in multiple ways. Below are several examples:

- A request (e.g. SMPT?) asks server for the URL of a webpage that describes the device, given ProductType (e.g. LED_Light_Socket), ProductSKU, and VenderID.

- A request asks for a diagnostic string (e.g. "Device OK", or "Device Temperature Sensor Broken") given ProductType, ProductID, VenderID, and string from device (e.g. with diagnostic voltages, etc.).
- A request asks for URL of distributor (e.g. amazon.com) webpage for valid replacement module.
- A request asks for replacement module be shipped to building owner after noticing it is broken. The building owner might not know something broke until they receive the Amazon package and read the enclosed note explaining where one needs to place the contents of the box.

There is something called "end-to-end plug-and-play" which is where you plug in a device, within a building, and it automatically connects to an outside server without requiring setup. This is a bit tricky for a variety of reasons, which is beyond the scope of this document. However, researchers can explore connecting to an outside server in an acceptable manner.

21.14Security

In theory, "The Russians", or someone, might want to screw with the network in time of conflict (or, we are the ones screwing with someone else). For example, someone might write to eeprom and damage your network (or perhaps all networks). Below are things one might do to improve security:

- Disallow software update via network.
- Disallow programming Socket memory IC after it had already been programmed, unless one has physical cable that attaches socket to tablet computer.

22 Network Design

22.1 Libraries of ImmutableData and ImmutableStrings

Libraries of ImmutableData and ImmutableStrings capsules (arrays of structs) are copied throughout the network to help other devices and controllers understand the system. If each DeviceCommon ImmutableData capsule is 100bytes and you have 1000devices in your building, then you might end up with 100KB (100*1000) on your smartphone to help efficiently interact with the network.

If there is a request for immutable information, the controller does not go to the device, and instead returns information from its library. Same with the Automation Box Master Controller (AMC) that supports 1 to 14 BuildingBus networks. If one has a smartphone that interacts with the system, it might also contain a library of immutable information. Databases reduce network traffic significantly.

We use the term "library", yet we are not referring to formal SQL servers. Instead, one might pack C structs into a big block of memory, and then have arrays of offsets that tell you where these are located. Then, given a physical address to a device and port, they could get a copy of that struct.

One can put together a list of device serial numbers, sort it, and use it to find the information associated with that device (i.e. build searchable index).

If one adds a device to the library, they might recreate searchable indices and store to flash memory, yet not commit to the new version until they complete the Add operation. This would resist corruption due to losing power or due to a memory error during the Add operation (i.e. abort add operation and revert to original database if it does not fully complete).

One might also have the microcontroller measure the power supply voltage in the background (e.g. every 10msec) and if it drops, indicating powering off, then implement an emergency shutdown procedure that resists damage to stored data.

Instead of asking the network for the value of each field, one at a time, it would be more efficient to copy an entire library at one time. For example, if the AMC has all immutable data in a 100KB file, and one interacts w/ the AMC via high speed Ethernet, then it would be most efficient to read the entire 100KB file into the client computer at one time (e.g. into smartphone).

A BuildingBus device that maintains a sub-network (e.g. Window Network Controller) might maintain a library of immutable information for all devices on the sub-network (structs are packed into one binary blocs with arrays of offsets to find each). If the 1st 8bit word of each struct is struct size, then the library software could treat each as a binary block and store it in a way that facilitates finding it later. Also, the library might have a 32bit DatabaseVersion which is incremented each time the database is changed in any way (e.g. struct is added). Then, higher level network components can check that version, and if different from what they have, upload a copy of the entire database. Subsequently, the BuildingBus Controller could gather information that pertain to all sub-networks, and the AMC in turn could gather all BuildingBus Controller libraries. And your smartphone might occasionally copy the entire AMC library in one block transfer (e.g. move 300KB file, 300 devices, 1KB each), enabling smartphone software to work with devices without placing a burden on the network.

CANopen is an example of software that maintains CANbus devices via a database; however, it was designed for a different kind of network, and therefore might only be marginally helpful. For details, search "CANopen" and "MicroCANopen" in file Ma2_IOT; and for more ideas, search for "Device Object Model Research" in file Ma2_IOT.

22.2 BuildingBus Memory Sockets and CANbus ID

Recall that devices plug into physical BuildingBus sockets; and the physical electrician who wires the physical building programs the sockets with things like physical distance to floor, physical location, socket type, unique 64bit serial number, and Device CANbus ID on BuildingBus network (DevicePhysicalAddress #0…31). Or, the physical window manufacturer wires window sub-network and programs WindowBus sockets with Device CANbus ID on BuildingBus SUB-network (SubNetworkPhysicalAddress #0…7). As noted previously, this facilitates Replace and Run, which entails home owner replacing a device, without engaging in setup. Also, this means one talks to sockets, not devices, metaphorically.

Participants (e.g. Smartphone, BuildingBus controller) store socket information in a library, and build search indices that do things like match socket serial number to device information (i.e. application software might refer to device with socket serial number since it is less likely to change).

CANbus ID's are sort of like the physical address to your house (e.g. 15 Apple Ave.). They are not expected to change. These could be programmed into a socket when the network is built (e.g. electrician wires building or window manufacturer physically builds sub-network); or Devices could, in theory, ask the Network Controller for their socket's CANbus ID (also called "DevicePhysicalAddress") when they first power up and join the network (perhaps done once in a socket's lifetime).

22.3 Electrician Programs Sockets via Tablet Computer and Electrical Drawing Software

Currently, power electronics engineers create building electrical wiring diagrams and these tell electricians how to wire a building. It is likely these power engineers would add BuildingBus sockets to their drawings since they are the ones designing the system (not the electrician, who installs what is specified by the engineer). Electrical Diagramming software is likely to support having the engineer define things like a location text (e.g. "living front wall vent"), vertical distance from floor to socket, electrical labels (e.g. electrical boxes might be labeled d EB1, EB2, etc. on drawing), and DevicePhysicalAddress (unique 1..31 CANbus ID for each socket on network). Later, when electrician wires building, they might connect a tablet computer to socket memory IC via cable, and tap location on drawing to program memory IC. This programming would be done once when the building is first wired, and later, when modules are plugged in (and replaced), no setup would be required.

23 Moving Data from One Device to Any Other Device

23.1 AddressIdentifiers Facilitates Routing of Data Packets within BuildingBus System

If we have up to 16 modules in the card cage (4bits), up to 31 devices on a BuildingBus network (5bits), up to 15 devices on a sub-network (4bits) and up to 8 ports (3bits); then one could identify a position (i.e. specific port) on the network with a 16bit AddressIdentifier (4+5+4+3 = 16bits = 2bytes), for example. One could then use this to route a data packet to a specific position in the network. Also, one could have a 2byte ReturnAddressIdentifier (RAI) that tells the device where to send requested information.

In the Network Topology diagram, shown above, one can see there are 4 levels of network topology: BB Controller (0...15, ControllerPhysicalAddress), BB Network (0...31, DevicePhysicalAddress), Sub-Network (0…15, SubNetworkPhysicalAddress), and Port (0...7, PortIndex). In all cases, the upstream controlling element is given address #0. In other words, the AMC is ControllerPhysicalAddress#0 at the BB Controller level, The BuildingBus Controller is at DevicePhysicalAddress = 0 on the BuildingBus 2-wire network, the sub-network controller (e.g. Window Master Controller module) is at SubNetworkPhysicalAddress = 0 on the sub-network, and the device itself is at PortIndex = 0 within the device software.

One might store the 2byte outgoing TargetAddressIdentifier in the 29bit MessageID field. And might store the 2byte ReturnAddressIdentifier in a data field when requesting information.

Below are addresses of devices at each network level:

Component	CtlrPhysAddr	DevicePhysAddr	SubNetworkAddr	
AMC	0	ignored	ignored	
Network Controller	1…14	0	ignored	
Network Device	1…14	1…30	0	Subnet Controller or No Subnet
Subnet Device	1…14	1…30	1…14	

Researchers need to think about the different possibilities and put together something that works well.

23.2 MasterCommand

When one sends a CANbus message, they are doing one of four things which is specified in a 2bit MasterCommand field.

- 0b01 FireAndForgetCmd — Initiator is asking Target to perform a function and NOT return data (and not send error information if error occurs).
- 0b10 FunctionCallCmd — Initiator is asking Target to perform a function, and return error information per ErrorPolicy possibly return data.
- 0b00 ReturningResponseData — Target is responding to a FunctionCallCmd by sending a 0 to 8byte response back to Initiator, and possibly including an error code.
- 0b11 BlockTransferOneSegment — Transmitting one segment that is part of a long data block.

These 2bits are placed at the beginning of the CANbus 29bit identifier field; and they influence priority since CANbus gives more priority to the lower value. ReturningResponseData has highest priority since we have already invested in setting up the transfer and want it to end as quickly as possible. BlockTransferOneSegment has lowest priority since transferring many segments will take much time no matter what and we want others to chat while that is occurring.

23.3 FunctionCode

If doing FunctionCallCmd then the target will return some information, and if doing FireAndForgetCmd then no information is returned. The 0…31 FunctionCode (5bits) specifies which function is to be performed. CANbus messages are sent to Ports (e.g. temperature sensor, motor, device itself); therefore, the FunctionCode is specific to each PortType (e.g. FunctionCode = 3 sent to a temperature sensor might request accuracy of sensor; whereas FunctionCode = 3 sent to a motor might tell motor to turn off). If you have 1 to 31 unique FunctionCodes for each type of port (2byte PortType), and 65K different ports, then the system could potentially support 2M different functions (31*65K), in theory. 31 functions is a lot of functions for one type of port. If you want more, consider input parameters that specify more features.

23.4 Standard Int16 Read/Write Value

Each PortType and each DeviceType supports reading and write a 16bit integer value that relates to what they do. For example, a temperature sensor might return a temperature in 0.01°C units. Motors attached to curtains, thermal covers, blinds, open/close window might have a 0 to 32500 value correspond to percent open (i.e. 0 is 0% open, 32500 is 100% open). Lights might interpret 0 to 32500 as 0 to 100% illumination. On/off valves, on/off switches, and on/off motors might consider 0 to be off, and 1 on. Analog sensors could work with -32500 to +32500 values, with agreed upon units (e.g. 0.01°C with temperature).

If one controls a device by writing a value, they could also read the device to get status of the actuator (e.g. set LED to 50% illumination via 16250 write, read it back and see 16250 if the LED is illuminated 50%).

The high 267 values (+32501 to +32768) of the int16 might be set aside for error codes.

Working with standardized int16 values helps automation/control software interact w/ the system without knowing too much about specific devices and ports.

23.5 BuildingBus Talks to Others via IP (IPsocketID)

If one moves outside the 1 to 14 module BuildingBus System, to a network location beyond our 2byte AddressIdentifier (e.g. AMC Automation Box talks to Smartphone via Ethernet), then we are probably working with 4byte or 16byte IP, and are looking at faster and larger data packets. To implement, the AMC might open a socket to an external node via IP (e.g. connect to smartphone) and assign this an internal IPsocketID.

Also, it is possible there is no AMC and instead one BuildingBus Controller connects directly to Ethernet (or Wi-Fi). Subsequently, the BuildingBus Controller would also be the AMC and open a network socket and assign a IPsocketID. Note that the Xmc4300 and Xmc4800 provides Ethernet (Xmc4200 is similar yet no Ethernet). When sending to an external node, the 16bit ReturnAddressIdentifier might specify AMC processor in the most significant 4bits (ControllerPhysicalAddress = 0), set one bit that specifies external IP node, and then specify IPsocketID in lower 11bits (4+1+11=16).

23.6 Keeping track of each CANbus Session with an 8bit MsgSessionID

When an initiator issues a FunctionCallCmd command, they need to include a 2byte ReturnAddressIdentifier in the outgoing message so the target knows where to direct the response data. Keep in mind that any device on the network can ask any other device for information, not just the AMC. When the initiator receives response data it needs to match it to the original request, since the initiator might have multiple requests outstanding and needs to match the returned data w/ the original request. It might do this with a 1byte 0 to 255 field (or possibly larger) that contains an MsgSessionID, which increments each time a FunctionCallCmd request is initiated.

The initiator maintains a MessageSession struct with information on the operation and this is placed into an array (e.g. 256 MessageSession structs), and the MsgSessionID is an index into this array. The MsgSessionID is sent to the target and is returned along with the response data. Subsequently, when the request comes back, the initiator knows which request it is associated with.

Some requests originate from an external IP address (e.g. AMC attached to Ethernet), in which case the Message Session struct would contain information that relates to an Ethernet socket (i.e. refer to an IPsocketID).

If the Target Address is the AMC (ControllerPhysicalAddress = 0), then one can identify this with the first 4bits of the AddressIdentifier, which leaves 12bits in the AddressIdentifier for other things (e.g. for MsgSessionID, IPsocketID, StreamSegmentIndex).

A 1byte MsgSessionID is one way of matching FunctionCallCmd requests with response data, yet there are others that might be considered by researchers. Many of the processors on the system do not have much RAM memory; therefore, one needs to be careful with how they approach this.

If it takes 3mSec to send a message and 3mSec to get a response, and messages are required to respond within 2 seconds before they timeout, then one might have a maximum of 333 different messages running at one time within one BuildingBus 2-wire network (2/(0.003*2)), in theory.

If an initiator is doing FireAndForgetCmd, they might not maintain a Message Session struct (e.g. MsgSessionID = 0), since they are not expecting a response (even an error code).

A simple module might not do too many FunctionCallCmd requests at one time, and therefore might only maintain a small number of MessageSession structs (e.g. 16). However, the AMC processor might have 100-fold more RAM and might maintain 256 of these for each of 14 BuildingBus networks (e.g. 256*14*32=120KB of ram given 32byte structs). If this is the case, then the BuildingBus network controller would need to include the 4bit ControllerPhysicalAddress value w/ CANbus messages sent to the AMC, enabling the AMC to identify the correct session struct (which has information about the request).

23.7 FireAndForgetCtr

When one sends a FireAndForget command, they also include a 0…63 FireAndForgetCtr value (6bit). This increments each time an Initiator sends a FireAndForget command and this helps avoid CANbus collisions by making it less likely for two nodes to send the exact same identifier to the same target at the same time (it is still possible, yet 64:1 less likely).

23.8 ErrorPolicy

The 2bit ErrorPolicy parameters specifies how errors are handled (0…3) and is used with the FunctionCallCmd command. The FireAndForget command does not have an error policy since it does not respond to errors in any way. If one wants error handling, they need FunctionCallCmd. Below are several ErrorPolicy options:

- IfThereIsNoErrorThenDoNotIncludeErrorValue
- AlwaysRespondWith2byteErrorCodeEvenIfThereIsNoResponseDataAndNoError -- this is sort of like Acknowledge message
- IfThereIsErrorThenAppend2byteErrorCodeToRespondDataAndReturnToInitiator
- IfThereIsErrorThenAppend2byteErrorCodeToRespondDataAndReturnToInitiatorAndAlsoSendErrorReportToAMC

If one is responding to an error condition, they might want to communicate via FireAndForget; otherwise, they risk loading the network with lots of errors.

If an Initiator wants to receive an acknowledgement after sending a command, ErrorPolicy would be set requesting an error code be returned, even if there is no error (i.e. errorCode = 0).

One of the error policies involves error codes also being sent to the AMC, which in turn could push them into a circular buffer and maintain error counters (increment each error). This would help developers better understand their network.

23.9 Transmitting Long Blocks of Data via Multiple CANbus Messages (SegmentIndex)

If one needs to transmit more than 8bytes, then they need to segment the long data block into multiple messages. A StreamID refers to the entire block of data, and a SegmentIndex increments each time one sends a CANbus message with 1 to 8bytes of data. Here is an example. Initiator Device sends FunctionCode to Target Device that request a 130 byte immutable DeviceCommon ImmutableData capsule struct be transferred from Target to Initiator. A StreamID is included in this original request. The Target returns multiple CANbus messages where the 29bit identifier field is loaded with: MajorCommand = BlockTransferOneSegment (2bit), TargetAddressIdentifier (16bits), StreamID (4bits) and SegmentIndex (7bits). The 8 data byte payload field would then be loaded with data. For example, 17 frames would be needed to transfer 130 bytes.

If one sends a command to device PortIndex=0 (i.e. the device itself) and FunctionCode specifies that the target return the DeviceCommon ImmutableData, then it would respond w/ a block transfer, perhaps 50 to 300 bytes, as described above.

A 7bit SegmentIndex would support 127 different segments (1 for header), each 8bytes, which works out to a 1016 byte data block (127 x 8); and a 4bit StreamID would support 16 different block transfers at one time.

Before sending the segments, one must establish the connection between the two endpoints and make sure all parties along the way have the following information: StreamID, TargetAddressIdentifier and ReturnAddressIdentifier. This information would be used to coordinate the parties in the event of a bit error, which leads to resending missing data. Also, the destination needs to keep track of which segments are received, and request that missing segments be resent.

Lower 29bit Identifier values get priority over higher, according to the CANbus protocol; therefore, we position the 2bit MajorCommand before the 4bit ControllerPhysicalAddress field and set BlockTransferOneSegment to a high number to give it lower priority (0b11). When transferring a large

block via many messages, one does not want to stop single transfer activity, since many data stream messages will consume much time no matter what one does, and signal transfer might be time critical.

For an example of data stream handling, search "ISO 15765" in file Ma2_IOT.

23.10Learning CANbus

To learn more about CANbus, one might consider the Natale, the Voss or the Pfeiffer CANbus books. The Voss book is an easy read. Also, one can read the Infineon CANbus controller manual, and search for "CANbus Summary" in Ma2_IOT.

The Xmc4200 processor supports CANbus 2b (≤ 8bytes/packet, ≤ 1M bps), yet not CANbus FD (≤ 64bytes/packet, ≤ 15M bps).

24 Broadcast and Groups

24.1 Broadcast

Broadcast involves sending one message to multiple devices. Each CANbus device can listen for its CANbus ID, and also listen for messages that are being sent to multiple devices. One might allocate the following addresses to refer to broadcast: ControllerPhysicalAddress = 15, DevicePhysicalAddress = 31, SubNetworkPhysicalAddress = 15. Subsequently, one would have 14 BuildingBus controllers (not 15), 30 maximum devices on a BuildingBus network (not 31), and 14 maximum devices on a Sub-Network (not 15). Also, recall that ControllerPhysicalAddress = 0 refers to the AMC, DevicePhysicalAddress = 0 refers to the BuildingBus Controller, and SubNetworkPhysicalAddress = 0 refers to the sub-network controller on the sub-network. Below are different sets of devices that can potentially receive a broadcast:

CtlrPhysAddr	DevicePhysAddr	SubnetPhysAddr	Devices That Receive Broadcast			
C = 0	D = ignored	S = ignored	AMC (no broadcast, this is message to AMC)			
C = 15	D = 31	S = 15	AMC,	All Network Controllers,	All devices in all networks,	All Subnet Devices
C = 15	D = 31	S = 0	AMC,	All Network Controllers,	All devices in all networks,	yet no Subnet devices
C = 15	D = 0	S = 0	AMC,	All Network Controllers,	yet nothing lower,	
C = 1…14	D = 31	S = 15		One Network Controller,	All devices on One Network,	All Subnet Devices
C = 1…14	D = 31	S = 0		One Network Controller,	All devices on One Network,	yet no Subnet devices
C = 1…14	D = 1…30	S = 15			One device in One Network,	All Subnet Devices

Broadcast is supported by all MasterCommands, including FireAndForgetCmd, FunctionCallCmd, ReturningResponseData, and BlockTransferOneSegment.

When one broadcasts, they can optionally specify a DeviceTypeKey in the payload area. This instructs devices to only respond if their DeviceType exactly matches the transmitted DeviceTypeKey. Also, one can optionally specify a PortTypeKey, where devices only respond if at least one of their ports exactly matches that value.

24.2 BroadcastControl

When one broadcasts, the 3bit PortIndex field is replaced with a 3bit BroadcastControl field. This contains bits that specify which broadcast fields have been appended to the payload buffer to describe the broadcast (if one exceeds 8byte limit they incur an error):

- EnableBroadcastField_GroupControlKey 7bit GroupNumber and 1bit PortIndex 1byte
- EnableBroadcastField_PortTypeKey PortTypeKey 2bytes
- EnableBroadcastField_DeviceTypeKey DeviceTypeKey 2bytes

If one enables multiple keys (we search for exact match to key), then one must meet multiple requirements in order for a device to accept the message.

If a PortTypeKey is specified, the device accepts the message if it contains a PortType that exactly matches the key; and if so, it directs the message to its first port that matches that type; otherwise it directs message to PortIndex = 0 or 1 as specified by the 1bit PortIndexKey field.

The GroupNumberControl field contains a 7bit GroupNumberKey and a 1bit PortIndexKey. The 1bit PortIndexKey directs the message to PortIndex#0 or #1 (assuming PortTypeKey is not included). The GroupNumberKey (0…127) specifies a group and the device only receives the message if it is a member of that group (which was set up previously).

If no keys are specified (i.e. BroadcastControl = 0b000), then all devices accept the message, and it is directed to the DeviceCommon port (portIndex = 0).

Broadcast data is packed into the CANbus message as follows:

29bit identifier:	d2	EnableBroadcastField_GroupControlKey,	1 enables GroupNumberKey/PortIndexKey
	d1	EnableBroadcastField_PortTypeKey,	1 enables PortTypeKey
	d0	EnableBroadcastField_DeviceTypeKey,	1 enables DeviceTypeKey

Data Payload:	..	d7…d1	7bits	GroupNumberKey (0…127 value)
		d0	1bits	PortIndexKey = 0 or 1
	..	PortTypeKey, LS Byte		
	..	PortTypeKey, MS Byte		
	..	DeviceTypeKey, LS Byte		
	..	DeviceTypeKey, MS Byte		

24.3 Groups

Devices can be set up as a member of a group, as noted previously. An example is you have 15 LED lights in your living room, they are members of group N, and a broadcast command with GroupNumber = 15 would sets the illumination level of all at one time. A wall light switch might be set up to send a broadcast command to the group when changed.

24.4 Special Groups Common to All Devices

There are several pre-determined groups, summarized below. If key fields are not included, the message is directed to the DeviceCommon port (i.e. PortIndex = 0).

- GroupNumber_AllDevicesInSystem (GroupNumber = 1)
  All devices in system are members of this group. Messages moves through the system as specified in the AddressIndentifier field.
- GroupNumber_AllDevicesOnCANbusWire (GroupNumber = 2)
  Similar to the above, yet message does not move through gateways and instead is local to one CANbus cable. This is helpful if the system is first starting up and devices/gateways may not know their address in the system. To send a message to all devices on CANbus cable even if none of them know anything about who they are, transmit to: Network = 15, Device = 31, Subnet = 15, GroupNumber = 2, PortIndex = 0. This looks like it is going to all devices in the system, yet broadcast to GroupNumber = 2 is a special case, and is not repeated at gateways.
- GroupNumber_TheControllerOnCANbusWire (GroupNumber = 3)
  This is similar to above, except only controller for the CANbus cable is to accept this message. For example, if you are a device on a network and you broadcast to this group, your network controller should accept. Or, if you are a subnetwork device, your subnet controller should accept. This messages does not pass through gateways (similar to GroupNumber #2). To send a message to the controller on your CANbus cable even if you have no idea of your address, transmit to: Network = 15, Device = 31, Subnet = 15, GroupNumber = 3.

The DeviceCommon port (included in all devices, PortIndex = 0); makes uses of Groups 1, 2, and 3; to set up, test, and maintain devices. DeviceCommon responds to specific functions, as defined by the FunctionCode parameter.

24.5 Working With Devices Who Have Not Yet Been Programmed with an Address

If a device has not yet been programmed with a 3-value Address (network #, device #, subnet#), then one can still talk to it via Broadcast to Groups 1, 2, 3; described above. These provides features that enable devices to request, and receive, set up information.

24.6 Free and Open Code

The follow constructs supports broadcast:

- `struct BuildingBus_BroadcastDescriptor:` Struct that describes a broadcast.
- `LOAD_BroadcastDescriptor ():` Loads above struct w/ DeviceTypeKey, PortTypeKey, GroupNumberKey, etc.
- `Extract_BroadcastDescriptor_Given_CANbusMsg ():` Extracts parameters given a CANbus message.
- `enum BuildingBus_BroadcastType:` List of different areas of network one can broadcast.

25 Packing BuildingBus Fields into CANbus Msg 8bytes + 11/29bits

CANbus frames contain 0 to 8 data bytes along with a CANbus identifier field that is either 11bits or 29bits.

If two CANbus messages are transmitted at the same time, the lowest valued 29bit identifier will receive priority and continue transmitting. If two messages transmit at the same time and the identifier fields of the two transmissions are identical (yet data payload is different), then a "CANbus collision" occurs. Dealing with this is beyond the scope of this document; however, in summary, we want to avoid.

Parameters are packed into messages in multiple ways, depending on how they are used. Here is a list of common parameters: TargetAddressIdentifier (16bits), ReturnAddressIdentifier (16bits), MsgSessionID (5 or 8bits), StreamID (4bits), SegmentIndex (7bits), NumOfResponseDataBytes (3bits), MasterCommand (2bits), FunctionCode (5bits), FireAndForgetCtr (6bits), and 0 to 7bytes of qualifying data.

It is likely one would pack these parameters into 0 to 8 data bytes plus a 11 or 29bit identifier field using multiple techniques, depending on the situation. Here are several that might make sense:

25.1 FireAndForgetCmd -- Transmit FunctionCode and 0…7bytes qualify data, no response

We transmit a command yet in no circumstances is data returned, even if an error is incurred. ReturnAddressIdentifier and MsgSessionID is not included; subsequently, we have no way to contact the Initiator, if we wanted to. If you are not comfortable with this, then consider FunctionCallCmd.

29bit identifier:	2	MasterCommand = FireAndForgetCmd	2+16+6+5=29
	16	TargetAddressIdentifier	
	6	FireAndForgetCtr (0…63 value)	
	5	FunctionCode (0...31 value)	
Data Payload:	0 to 8 bytes of qualifying data		

25.2 FunctionCallCmd -- Similar to above, yet supports ErrorPolicy and Response Data

The 29bit MessageID field is loaded with the following when sending a FunctionCallCmd: MasterCommand (2), TargetAddressIdentifier (16), MsgSessionID_Small (4), and ErrorPolicy (2). Then, in the 8byte data payload area we pack in the following order: Optional MsgSessionID_Big (8), Optional 2byte ReturnAddressIdentifier, and 0 to 6 bytes of qualifying data. If one exceeds a total of 8 payload bytes, we incur an error.

29bit identifier:	2	MasterCommand = FunctionCallCmd	2+16+8+3=29
	16	TargetAddressIdentifier	
	4	MsgSessionID_SmallField, 0…15 value, placed in identifier to discourage collision) If MsgSessionID_SmallField = 0, then place MsgSessionID at payload data [0] If MsgSessionID_SmallField = 1, then place MsgSessionID at payload data [0] & Request Responder Address	

	2	ErrorPolicy (0...2value)
	5	FunctionCode (0…31 value)
Data Payload:	..	Possible MsgSessionID_BigField, 2...255 value, enabled if SmallField ≤ 1
	..	Possible ReturnAddressIdentifier, LS Byte
	..	Possible ReturnAddressIdentifier, MS Byte
	..	0 to 6 bytes of qualifying data, e.g. function input parameters

25.3 BlockTransferOneSegment -- Transmit one 1…8byte Segment within long data block

The data stream is set up with an initial FunctionCallCmd where ErrorPolicy is set, along w/ other parameters (e.g. total # of bytes to transfer, information on what data is to be sent). This is followed by multiple BlockTransferOneSegment messages, each of which transfer 1...8 bytes.

29bit identifier:	2	MasterCommand = BlockTransferOneSegment	2+16+5+6=29
	16	TargetAddressIdentifier	
	4	StreamID (0…15 value, placed in identifier to discourage collision)	
	7	SegmentIndex (0...127 value)	
Data Payload:	...	1 to 8 bytes of segment data	

25.4 ReturningResponseData -- Target is sending response data back to Initiator

After receiving a FunctionCallCmd, the Target might return information back toward the Initiator. This includes regular data (e.g. function output parameters) and possible error information.

29bit identifier:	2	MasterCommand = ReturningResponseData	2+16+5+6=29
	16	TargetAddressIdentifier	
	8	MsgSessionID (0…255 value, placed in identifier to reduce collision)	
	1	ErrorOccurred (0 if no error occurred, 1 if error)	
	1	Returning2byteErrorCodeAfterResponseData (if set, 2byte error code follows returned data)	
	1	Returning2byteResponderAddressAfterErrorCode	
Data Payload:	...	0 to 8 bytes of response data, e.g. function output parameters	
	..	Possible 2byte ErrorCode, LS Byte	
	..	Possible 2byte ErrorCode, MS Byte	
	..	Possible 2byte ResponderAddress, LS Byte	
	..	Possible 2byte ResponderAddress, MS Byte	

25.5 Example CANbus Messages

Here are some examples.

- One sends FireAndForgetCmd command with 1byte qualifier to physical curtain to opens it a variable amount with a 0 to 255 value. Or, we do the same yet with 2byte qualifier, where 2nd byte tells curtain over how many seconds to open (e.g. open slowly over 10 seconds, for a special effect). In the second example, the 29bit MsgIdentifier field is loaded with MasterCommand = FireAndForgetCmd (2), TargetAddressIdentifier (16), FireAndForgetCtr (6), and FunctionCode (5); and the data payload is loaded with 127 (to specify 50% open), and 10 (# of seconds).
- One sends FunctionCallCmd command to temperature sensor with no qualifying bytes and return datasize is specified as 2bytes. In this case, the 29bit MsgIdentifier field is loaded with MasterCommand = FunctionCallCmd (2), TargetAddressIdentifier (16), MsgSessionID (8), and ErrorPolicy (3); and the data payload is loaded with FunctionCode (5), NumOfResponseDataBytes (3), ReturnAddressIdentifier LS byte, and ReturnAddressIdentifier MS byte. Device would return a 2byte int16 value, perhaps in units of 0.01C.
- Or, do the same as above yet FunctionCode says we are requesting that the measurement be the average temperature of multiple readings over one power-line cycle (e.g. 1/60th or 1/50th of a second), to reduce noise.
- Or, do the same as above yet FunctionCode says the port is to return things like: absolute accuracy of the sensor (e.g. in 0.002°C units, uint16, 0 to 12°C range) and noise of the sensor (e.g. in 0.002°C RMS units, 2bytes). In this case, the port might access data that resides in its immutable ImmutableData struct.

Some temperature sensors are only accurate to ±3°C whereas others are good to ±0.1°C. Noise sometimes decreases 10-fold when one averages for one power-line cycle. Temperature between floor and ceiling is sometimes 2°C. Subsequently, one must deal with these issues if doing temperature control and desire satisfied occupants. Also, it is interesting to note that many of the existing networking protocols ignore sensor accuracy, integration vs noise, and sensor location.

26 Devices interact with Ports via Functions

As noted previously, one device communicates with another device(s) via: 0 to 5 output ("qualifying") bytes, 0 to 8 response bytes, and a FunctionCode code. From a code point of view, this is like a function with both input and output parameters. Each of these is directed to a specific type of port. An example function might look like the following within the initiator's code:

```
TemperatureSensor_Measure_Temperature_10mC_units (&targetAddrIdentifier, &inputParameters, &outputParameters);
```

26.1 Researchers define Functions in a Spreadsheet

One might define each function in a spreadsheet, one row per function, with the following columns:

- FunctionName
- PortType (16bit value)
- PortType_Name

- FunctionCallCmd or FireAndForgetCmd
- FunctionCode_Value
- Set of Input Parameters 1..N: Type, Name, SizeBytes, Comment
- Set of Output Parameters 1..N: Type, Name, SizeBytes, Comment

Each parameter is given their own column, and each function their own row. To support 4 input parameters, one might have 16 columns: In1_Type, In1_Name, In1_SizeBytes, In1_Comment, In1_Type, In2_Name, In2_SizeBytes, In2_Comment, etc. To support 4 output parameters, one might have another 16.

One then exports the spreadsheet to a CSV file and loads with a computer program that iterates through the rows and generates .txt (or .html) reports along with .h/.c code. The spreadsheet would include all Functions used with all PortTypes in the entire system; therefore, the generated code would relate to all types of ports, both from the perspective of the Initiator (Client) and Target (Server).

Here's an example. One creates a spreadsheet and adds one row with the following parameters.

- FunctionName: GetTemperature_16mSec_10mC_units
- PortType: 0x0563
- PortType_Name: TemperatureSensor
- CommandType: FunctionCallCmd
- FunctionCode: 0x11
- Input Parameter #1 Type, Name, SizeBytes, Comment: uint8, integrationTime_mSec_units, 1, amount of time temperature is averaged during measurement
- Output Parameter #1 Type, Name, SizeBytes, Comment: int16, temperature_10mC_units, 2, measured temperature in 0.01C units

26.2 Software Generates One Report for Each Function

Software then generates a .txt or .html report for each function, .h/.c files for initiator, and .h/.c files for target, one file for each PortType. Below is an example report for one function:

Function C Prototype

```
void TemperatureSensor_GetTemperature_16mSec_10mC_units (
    TargetIdentifier targetAddrIdentifier,
    TemperatureSensor_GetTemperature_16mSec_10mC_units_INPUT_PARAMETERS *inputParameters,
    TemperatureSensor_GetTemperature_16mSec_10mC_units_OUTPUT_PARAMETERS *outputParameters);
```

PortType

TemperatureSensor, PortType = 0x0563

Input Parameter Struct

```
// Input parameters for TemperatureSensor PortType, GetTemperature_16mSec_10mC_units function
typedef struct TemperatureSensor_GetTemperature_16mSec_10mC_units_INPUT_PARAMETERS
{
  int8_t integrationTime_mSec_units; // amount of time temperature is averaged during measurement in 1mSec units
}
TemperatureSensor_GetTemperature_16mSec_10mC_units_INPUT_PARAMETERS;
```

Output Parameter Struct

```
// Output parameters for TemperatureSensor PortType, GetTemperature_16mSec_10mC_units function
typedef struct TemperatureSensor_GetTemperature_16mSec_10mC_units_OUTPUT_PARAMETERS
```

```
{
  int16_t temperature_10mC_units; // measured temperature in 0.01C units
}
TemperatureSensor_GetTemperature_16mSec_10mC_units_OUTPUT_PARAMETERS;
```

Example Client Code

```
TemperatureSensor_GetTemperature_16mSec_10mC_units_INPUT_PARAMETERS inParams;
TemperatureSensor_GetTemperature_16mSec_10mC_units_OUTPUT_PARAMETERS outParams;

inParams.integrationTime_mSec_units = …; // amount of time temperature is averaged during measurement

TemperatureSensor_GetTemperature_16mSec_10mC_units (&targetAddrIdentifier, &inParams, &outParams);

… = outParams.temperature_10mC_units; // measured temperature in 0.01C units
```

Outgoing CANbus Msg

```
MasterCommand = MasterCommand_FunctionCallCmd;
NumOfResponseDataBytes = 2;
FunctionCode = 0x11; // GetTemperature_16mSec_10mC_units
TargetAddressIdentifier = ..; MsgSessionID = ..; ErrorPolicy = ..; FunctionCode = ..;
// MasterCommand (2), TargetAddressIdentifier (16), MsgSessionID (8), ErrorPolicy (3)
Identifier_29bits = (MasterCommand<<27) | (TargetAddressIdentifier << 11) | (MsgSessionID << 3) | (ErrorPolicy <<0);
Data#0:     (FunctionCode <<5) | (ErrorPolicy << 0)
Data#1:     ReturnAddressIdentifier Least Significant Byte
Data#2:     ReturnAddressIdentifier Most Significant Byte
```

C Code to Define Msg

```
void LoadMsgDescriptor_TemperatureSensor_GetTemperature_16mSec_10mC_units(
    /* OUT */ BuildingBus_CANbusMsgDescriptor                                *msgDescriptor,
    /* IN  */ BuildingBus_AddressIdentifier                                 *targetAddressIndentifier,
    /* IN  */ BuildingBus_AddressIdentifier                                 *returnAddressIndentifier,
    /* IN  */ BuildingBus_BroadcastDescriptor                               *broadcastDescriptor,
    /* IN  */ ErrorPolicy                                                   errorPolicy,
    /* IN  */ uint8_t                                                       msgSessionID,
    /* IN  */ TemperatureSensor_GetTemperature_16mSec_10mC_units_INPUT_PARAMETERS   *inParams)
{
    LOAD_CANbusMsgDescriptor_FunctionCallCmd(
        msgDescriptor,
        MasterCommand_FunctionCallCmd,                                      //  masterCommand
        TemperatureSensor_MinorCommand_Read16mSec_,                         //  minorCommand
        sizeof(TemperatureSensor_GetTemperature_16mSec_10mC_units_INPUT_PARAMETERS),    //  numOfQualifyingDat
        (uint8_t *) inParams,                                               //  qualifyingDataPtr
        errorPolicy,
        msgSessionID,
        sizeof(TemperatureSensor_GetTemperature_16mSec_10mC_units_OUTPUT_PARAMETERS),   //  numOfResponseDataB
        targetAddressIndentifier,
        returnAddressIndentifier,
        broadcastDescriptor);
}
```

26.3 Code Generation

The software that scans the spreadsheet might instantiate one object for each row in the spreadsheet (e.g. function), where class parameters correspond to spreadsheet columns. These could then be added to lists, one list per PortType. And then one could iterate through the lists to generate .h and .c code for each PortType. Also, the software could generate a master BuildingBus_Interface.h file that includes all the PortType .h files, along with other common information (e.g. enum with PortType codes).

26.4 Publically Accessible Website

In the future, we will probably have a publically available website where participants can register, get their own PortType codes (int32, 4e9 codes) and create their own functions with a web form user interface. The website could generate the .h/.c code in a manner similar to that done w/ the

spreadsheet. Also the spreadsheet could be loaded into a database which is connected to the website. In summary, the spreadsheet is likely to involve into a website/database based system.

27 Port structs and Device structs

27.1 Public Website Could Assist in Struct Development

In theory, capsules could be managed by a website based database, where each field of each struct is stored in a database record, in a manner similar to what one does with functions. The website could generate .h and .c code, both for initiator and target, as needed. Also, one could generate FunctionCallCmd commands that enabled an initiator to read each field within the PortRegisters/DeviceCommon PortRegisters struct (or get 6 bytes at a time).

If a code generation website was available to the public; and anyone was allowed to create ports and devices; then one would want 32bits for VenderID, StructType, PortType and DeviceType. Students might create 100K new ports a year. If one creates a PortType, it does not necessarily mean anyone else will notice. Only a relative few port types would be recognized by a formal standards body.

27.2 FunctionCall Commands Read/Write Individual Fields within structs

Researchers explore having FunctionCall commands access individual fields within structs (i.e. via a FieldIndex). Also, if data is immutable, controllers could intercept and retrieve a value from a database, instead of using the network. Exactly how this works is a bit tricky, since one might want to know the offset into the struct (# of bytes) and element SizeBytes. The web server that maintains the structs would know these values and could place them into #defines or hard code them into macros.

27.3 Capsule Header

All capsules have the same fields at the top (common fields), as defined by the `ImmutableData_HEADER` struct:

```
typedef volatile struct PortDescriptor_HEADER
{
    BB_StructPrefix_8byte   prefix_8byte;                 //  header at beginning of all structs

                                                          //  16bits  StructSize in bytes, 0..65K
                                                          //  32bits  StructType, unique code for each type of struct, does not
                                                          //  4bits   StructDesignRevision, programmer adds field to struct and
                                                          //  1bit    internal data is stored little or big endian
                                                          //  1bit    creator is end user or BuildingBus engineer
                                                          //  4bits   struct category (e.g. CommonDevice_DeviceDescriptor, PortS
                                                          //  3bits   key field set to 0x3

    uint32_t                supportedMinorCommands;       //  bit number X is set to 1 if MinorCommand X is supported by port, 0

    uint32_t                portType_uint32;              //  type of port
}
PortDescriptor_HEADER;
```

The uint32 supportedFunctionCodes field contains data that specifies which of the 0...31 FunctionCodes the port responds to. A value of 1 resides in bit location X of this field if FunctionCode X is supported.

27.4 Struct Header

The `BB_StructPrefix_8byte` struct is placed at the beginning of all structs (i.e. it is a header) to help coordinate the movement and storage of structs.

```
typedef volatile struct BB_StructPrefix_8byte              //  Header at the beginning of all structs

                                                           //  16bits  StructSize in bytes, 0..65K
                                                           //  32bits  StructType, unique code for each type of
                                                           //  4bits   StructDesignRevision, programmer adds fi
                                                           //  1bit    internal data is stored little or big en
                                                           //  1bit    creator is end user or BuildingBus engin
                                                           //  4bits   struct category (e.g. CommonDevice_Devic
                                                           //  3bits   key field set to 0x3
{
    BB_StructPrefix_Header2bytes _header;                  //  BbStructKeyAndFlags and BbStructCategory (same w

    uint16_t  _structSizeBytes_uint16;                     //  SizeBytes, 16bits, size struct in units of bytes

    uint32_t _BB_StructType_uint32;                        //  SizeType, 32bits, BbStructType enum, each BB str
}
BB_StructPrefix_8byte;
```

This header contains the following fields:

- 8bits StructSize_inUnitsOf4bytes, 4..1023byte struct size
- 32bits StructType, unique code for each type of struct, does not change when DesignRevision increments
- 4bits StructDesignRevision, programmer adds field to struct and *design* revision increases
- 1bit internal data is stored little or big endian
- 1bit creator is end user or BuildingBus engineer
- 4bit struct category (e.g. DeviceCommon ImmutableData)
- 3bits key field set to 0x3

28 BuildingBus Framework Software

BuildingBus C and C++ Framework software is currently being developed and partially implements constructs discussed above. There are two files types. One has a "BB" filename prefix and the other has a "MY" prefix. BB is developed by BuildingBus programmers and is not to be changed. "MY" is developed by industry programmers who create products. Most of the complexity is handled by the BB files, which reduces development time (and money). The framework supports simulation, and therefore one can much without hardware.

See Also:

- Voss's webpage with ~50 links to useful resources, including CANbus monitoring/debuggers\

28.1 Free and Open Code

Several structs help one navigate through the BuildingBus network:

- `BB_CANbusMsgDescriptor:` Describes one BuildingBus message and contains fields that correspond too many of the parameters described previously (e.g. ErrorPolicy, MsgSessionID, FunctionCode, MasterCommand, etc).
- `BB_AddressIdentifier:` Describes 16bit AddressDescriptor.

- `BB_CANbusMsgPacket:` Contains CANbus 29bit identifier and 8byte data payload.

Several functions help one pack and unpack CANbus message frames:

- `UNPACK_CANbusMessage ():` Finds BuildingBus fields (e.g. MsgSessionID, FunctionCode) packed inside a CANbus 29bit identifier and 8byte data payload buffer.
- `PACK_CANbusMessage ():` Given BuildingBus fields (e.g. MsgSessionID, FunctionCode), this function packs them into a CANbus 29bit identifier and 8byte data payload buffer.
- `LOAD_AddressIndentifier ():` Load AddressIdentifier struct (e.g. with controllerPhysicalAddress, devicePhysicalAddress, subNetworkPhysicalAddress, etc.)

Several functions help one define BuildingBus functions which are passed from one device to another:

- `CREATE_MsgPacket_FunctionCallCmd ():` Set up FunctionCall message.
- `CREATE_MsgPacket_FireAndForgetCmd ():` Set up FireAndForget message.
- `CREATE_MsgPacket_ReturingResponseDataCmd ():` Set up ReturingResponseData message.
- `CREATE_MsgPacket_BlockTransferOneSegmentCmd ():` Set up BlockTransferOneSegmentCmd message.

If you want to read from or write to any field within any capsule in any port on any device, then consider:

- `CREATE_MSG__Read_INT64_AnyField_AnyCapsule_AnyPort ():` Read from field.
- `CREATE_MSG__Write_INT64_to_AnyField_AnyCapsule_AnyPort ():` Write to field.

29 Further Networking Considerations

29.1 Addressing the AMC

When sending a CANbus message to the AMC, the TargetAddressIdentifier will set its ControllerPhysicalAddress field to 0 (AMC), while the other fields are ignored. Subsequently, when referring to the AMC, one might load the ignored bits within the target descriptor with *initiator* information. The advantage of doing this is twofold: (1) AMC might maintain a set of unique MsgSessionID's and set of unique StreamID's for each device and/or each network. (2) If the target to a FunctionCallCmd command is the AMC, then the AMC needs to know where to return the data.

- AddressIdentifier pointing to non-AMC device:

4	Target	0...14	ControllerPhysicalAddress	BuildingBus Controller ID
5	Target	0…30	DevicePhysicalAddress	Device CANbus ID on BuildingBus network
4	Target	0…14	SubNetworkPhysicalAddress	Device CANbus ID on SUB-network
3	Target	0…7	PortIndex	Port within device

AddressIdentifier pointing to AMC (only one AMC per system)

4	Target	0	ControllerPhysicalAddress = 0 refers to AMC	
4	*Initiator*	1...14	ControllerPhysicalAddress	BuildingBus Controller ID
5	*Initiator*	0...30	DevicePhysicalAddress	Device CANbus ID on BuildingBus network
3	*Initiator*	0…7	SubNetworkPhysicalAddress	ID on SUB-network -- only 3bits available

Notice the above scheme for the AMC target does not support Initiator SubNetworkPhysicalAddress 8...15, and does not provide Initiator PortIndex information. In cases where this is deficient, one might set the highest 12bits of the AddressDescriptor to all 1's, and place a complete 2byte ReturnAddressIdentifier in the 8byte data payload area.

One would need to make sure the above works ok with error frames.

29.2 Remote Frames

As noted by Voss CANbus book, CANbus remote frames sometimes have trouble, and should therefore possibly be avoided. If we did not avoid, then we could look at using them, provide Target was in a position to respond quickly. This would apply only to last leg on journey.

29.3 Error Frames

CANbus Error Frames are helpful at detecting errors. Also, additional BuildingBus messages are required to communicate error condition to the originator of journey.

29.4 Overload Frames

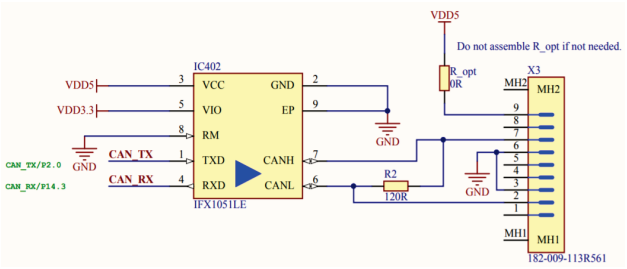
CANbus Overload Frames might be helpful in some way?

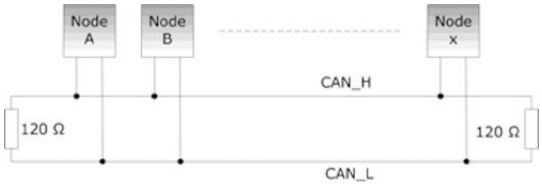
29.5 Endian

Xmc4200 processors and Intel Processors (e.g. Windows PC) are little-endian (LS Byte 1st); therefore, we transmit using little-endian. Also, free and open source code is set up to compile on little or big endian processors.

29.6 Connecting together two Xmc4200 PlatformToGo Boards via one CANbus Cable

If one wants to connect together two Xmc4200 PlatformToGo PCBs via CANbus, then a DB9-to-DB9 cable with four pin N to pin N wires (pins 2, 3, 6, and 7) will suffice. For details on how to program Xmc4200 CANbus, please refer to the Infineon MultiCAN User's Manual. Below is a schematic of the PlatformToGo CANbus interface. One would not want to connect together two independent power supplies (e.g. VDD5 at pin 9).





29.7 CANbus programming

A 29bit identifier is transmitted with each CANbus message. Each receiver uses a Mask to block out some of these bits, and then checks to see if the masked identifier (mask BitwiseAnd identifier) matches a value in its arbitration register. For example, if mask = 0xffff fffe (all bits received except bit d0), arbitration = 0xf0, and 29bit identifier = 0xf0, then the message will be accepted. For details, see MultiCAN User's Manual (AP32300) and MultiCAN presentation.

30 Comments on Physical Layer Signaling

Below are comments on how to electrically connect a window to a network, in no particular order.

RS-485 is a well-established standard that enables one to connect multiple devices along two wires; however, it has several issues:

- RS-485 transceiver IC's are designed to drive large capacitances on long cables at fast data rates via 60mA, which pushes devices to have larger power requirements and larger power supplies, which drives up cost.
- RS-485 requires daisy-chain wiring topology instead of the preferred tree topology which involves less wire and less installation time, and therefore less cost.
- RS-485 is often implemented with a microprocessor UART controller (inputs/outputs serial bytes), which needs constant attention from the microprocessor.
- RS-485 is almost always not fault protected against short circuit to 110/220VAC. This is risky when co-locating data wires with 110/220VAC power wires in a complex environment.
- One needs to route a common mode sense wire along with RS-485 data+- wires to establish a common mode voltage reference (wire connects together multiple transceiver IC ground pins). One might consider the earth ground wire, yet it sometimes includes excessive voltage drops due to flowing current. Alternatively, one might consider the 110/220VAC Neutral wire; however, it is sometimes swapped with Hot and it definitely involves flowing currents and voltage drops (which might exceed the transceiver IC common mode voltage requirement). One might consider isolated RS-485 transceiver IC's to improve common mode voltage tolerance, yet they cost money and they require a power supply on both sides of isolation. One might add a dedicated common mode voltage sense wire, yet this cost money and one needs to be careful to not place current on this wire and get a big voltage drop. RS-485 is a little tricky, yet works fine if one knows what they are doing. However, we would prefer something that works 99.999% of the time given unskilled installation personnel.
- If your power wires route to a full bridge rectifier (4 diodes), you lose your common mode sense voltage through the bridge when the power sine wave is < +-0.5V (all bridge diodes are off). If you have an isolated power supply and the secondary side COM wire is attached to transceiver IC gnd pin and also to your network common mode sense wire, then you will be ok, provided your sense wire does not include too much voltage drop.

- RS-485 often works well when coupled with DC power wires, provided that the Power- wire (which connects transceiver IC gnd pins) voltage drops are not high enough to exceed transceiver IC common mode voltage requirements.
- The typical way of providing media access control with RS-485 (i.e. deciding which device is allowed to talk) is to pass a virtual "token" between devices. This involves chatter between devices, it consumes time, it consumes processor cycles, and it is risky since the token can be lost and cause the network to stall. DALI is much slower than RS-485, yet when you look at overhead from token passing, it is sometimes not so slow.
- For details, search "RS-485 Electrical Signaling Protocol".

DALI provides two wire 16V/250mA control, mostly to lights; however, it has several issues:

- DALI 2 is slow at 1200 bps. However, it provides media access control (determine who is talking) and one command with dozens of bits is sufficient to move multiple devices (e.g. all lights within one group move from 20% illumination to 80% over 3 seconds).
- DALI interfaces are sometimes not protected against short circuit to 110/220VAC; however, this can be remedied with several additional components.
- DALI 2 optically isolates its Data+- wires from 110/220VAC power via an opto-coupler IC (e.g. #ELD217, $0.22, 2-channels) and a handful of parts. A low cost DALI transceiver IC would be helpful, yet is not available. Due to DALI's slow 1200 bps speed and multiple interface components, it is only popular with lights.
- DALI avoids common mode voltage problems since it signals on two 16V/250mA wires; and these are optically isolated from 110/220VAC power. One can swap the DALI+- signaling wires and swap the AC Neutral/Hot wires and it will still operate. This tolerance to wiring snafus reduces installation cost; which is terrific.
- For details, search "DALI (Digital Addressable Lighting Interface)".

CANbus is a popular 2-wiring signaling system; however, it has several issues:

- CANbus transceiver IC's are typically designed to drive 60Ω loads (120Ω resistor on each side of daisy-chain), which means they need lots of current (e.g. 60mA), which requires beefy device power supplies, which drives up cost. Shorting 120Ω to >12V power often results in damage.
- CANbus requires daisy-chain wiring topology instead of the preferred tree topology which involves less wire and less installation time, and therefore less cost.
- Many small microprocessors have an independent CANbus controller, which means they can implement media access control (decide who is talking), framing and moving of data while the microprocessor does other things; which is terrific. When working with RS-485, in comparison, the microprocessor UART controller (inputs/outputs serial bytes) needs constant attention from the microprocessor.
- CANbus is heavily used in the auto industry and is considered robust and fault tolerant.
- CANbus is similar to RS-485; however CANbus electrically defines what happens in a collision (two devices talk at same time) and RS-485 does not. With RS-485, the driver with the highest drive capability wins (most mA), whoever that is. With CANbus, the bus is at a logic "1" unless at

least one of the devices signals "0" (i.e. "wire-AND"). This known collision outcome enables CANbus to process collisions more easily.

One could implement building network devices at lower cost if the following standards existed:

- Two-wire signaling system that is optically isolated from 110/220VAC, supports wire-AND, supports tree topology, and runs faster than DALI 2 (e.g. BuildingBus AC).
- Two-wire voltage-sense signally system that is easily implemented with a low cost transceiver IC, supports tree topology, and supports wire-AND (e.g. BuildingBus 48V).

For more ideas, search this file for "WindowBus Physical Layer" and "BuildingBus Development Initiative".

31 Suggested Research Tasks

Below are several suggested tasks for researchers.

1. *AWDT1 - Mechanical Engineers Build Physical Prototype Rolled Thermal Cover (ME)*

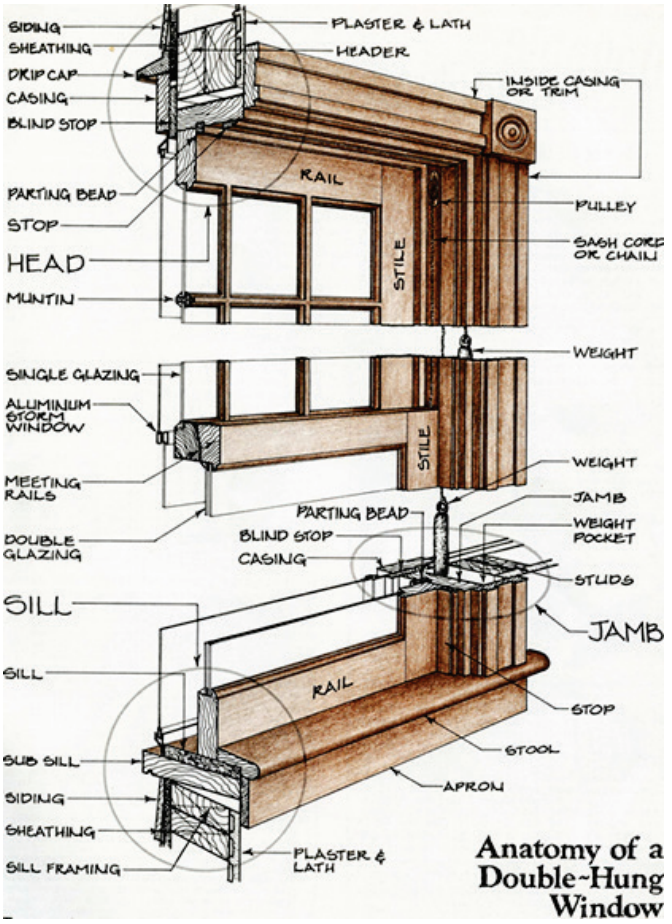
   - Mechanical engineers: place a vertical wood double hung window on a wheeled dolly, surround with 2x6 framing, place rolled thermal cover above widow "embedded" in "wall", seal sides of thermal cover against interior casing and sill (e.g. via rails to reduce airflow), interface to motor/gearbox, and connect motor and sensors to electronics prototype developed by other AWDT teams (i.e. Arduino/Click boards clipped together).
   - Propose mechanical standards for modules that contain motors, gears, PCB's. Support methods of accessing those modules for purposes of replacement. This allows building to survive over a long period of time without losing value.
   - Implement with 3D software, work with simulator and spreadsheet (e.g. thermal modeling), build prototype, test, publish report, and offer technology free-of-charge to multiple organizations (e.g. manufacturers of physical windows). All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.

2. *AWDT2 - Mechanical Engineers Build Physical Prototype of 1" to 2" thick Solid Foam Thermal Cover (ME)*

   - Similar to above, yet 1" to 2" thick solid foam panel instead of rolled. Panel includes two parts, one below and one above window since distance from floor to window bottom is often greater than window height.

3. *AWDT3 - Write Active Window Device Model software (EE or CS)*

- Develop Active Physical Window software models for multiple common networking protocols (e.g. DALI, KNX, and BACnet). The device reports to the network its list of sensors (and their accuracies) and list of actuators (and their power capabilities). Network then interacts with device requests (e.g. tell me outside temperature) and commands (e.g. move blind to 50% position).
- The first step is to accumulate information on current device models that involve physical windows or components to windows (e.g. motors, blinds, actuators, analog input, control output, temperature measurement), to learn more about the options. For more information, search "Active Physical Window Support".
- Programmers clip together existing Arduino/Click products as needed to test code.
- Write device software, write test software, test/debug, publish report, and offer software free-of-charge to multiple organizations (e.g. manufacturers of physical windows, manufacturers of processor reference designs).



- For details, search "Active Physical Window Support" and "Device Object Model Research Overview". All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.

4. *AWDT4 - Develop DALI based motor controller prototype that moves something up and down (EE or CS)*

- Using existing products (i.e. clip together DALI 2, motor controller and sensor click boards), build prototype of a DALI 2 device that tells the network it is an LED light (i.e. accepts 0 to 100% commands to set illumination level) and instead use illumination level commands to adjust deployment of something via a motor (e.g. string wrapped around motor shaft that lifts small weight at end of string).
- Add sensor hardware and software support. Consider click boards: humidity and temperature, weather monitoring (humidity, pressure, and temperature), environment monitoring (temperature, relative humidity, pressure and VOC), and pressure measurement.
- Clip together existing boards, write software, test (use existing DALI wall dimmer control to deploy small weight hanging from motor shaft), publish report, and offer all work free-of-charge

to multiple organizations (e.g. DALI Association, manufacturers of DALI hardware, manufacturers of physical windows).
- For details, search "Add Comprehensive Motor Support to DALI 2". All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.

5. *AWDT5 - Develop Physical Layer Electrical Signaling Standards for Buildings (EE)*

- Develop electrical signaling standard for BuildingBus 48V, BuildingBus AC, and WindowBus. For details, search for "Team A" in file Active Window Tasks.
- Work with simulator and spreadsheet, breadboard, produce Mikroe Click Board prototypes, write software that demonstrates board, publish report, and offer technology free-of-charge to multiple organizations (e.g. Mikroe, DALI Association).
- All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.

6. *AWDT6 - Write DALI 2 Software That Utilizes CANBUS Controller Hardware and CANBUS Data Layer instead of traditional DALI 16V/250mA network hardware and DALI frame (EE or CS)*

- Modify existing DALI 2 Master and DALI 2 Slave software to utilize CANbus data layer (framing and media access control) instead of the standard DALI 2 signaling protocol. Work with existing CANbus Click (instead of DALI Click).
- Utilize microprocessor CANbus controller hardware (i.e. pull against two 120Ω termination resistors) and sleep processor when not in use.
- Write software, test, publish report, and offer software free-of-charge to multiple organizations (e.g. DALI Association, manufacturers of DALI hardware).
- For details, search "Propose DALI 3 electrical signaling standard". All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.

7. *AWDT7 - Develop Tiny Flexible Power Supply Hardware (EE)*

- Develop tiny power supplies for microprocessors that are powered by 85...265VAC or 16...54VDC.
- Output 3.3V or 5V regulated power, 1 to 4mA while sleeping, 4 to 16mA while operating.
- Consider both isolated and non-isolated designs.
- Work with simulator and spreadsheet, breadboard, produce PCB, test, publish report, and offer designs free-of-charge to multiple companies (e.g. companies with buck converter reference designs such as TI and MPS).
- For details, search "Develop $1.50 Power Supply". All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.

8. *AWDT8 - Develop Protected DALI 2 and Protected RS-485 Hardware (EE)*

- Develop a DALI 2 Click board and an RS-485 Click board that is short circuit protected against 265VAC.

- Work with simulator and spreadsheet, breadboard, produce two PCB's, test, publish report, and offer design free-of-charge to multiple organizations (e.g. Mikroe, TI, RS-485 Transceiver companies, DALI Association, manufacturers of DALI hardware).
- For details, search "DALI Over-Current and Over-Voltage Protection" and "Design Current Limiting IC". All work is free and open, to encourage collaboration and adoption, to reduce worldwide energy consumption.