

USER'S GUIDE (DRAFT)

Mechanical, Electrical & Firmware Reference
(Applies to firmware versions 3.04 and later)

CAN-DO!

CREATED BY MEMBERS OF AMSAT-NA AND AMSAT-DL
SPONSORED BY AMSAT-DL

CAN-DO! User's Guide

Material provided by:
Bdale Garbee, KB0G
Chuck Green, N0ADI
Peter Guelzow, DB2OS
Lyle Johnson, KK7P
Karl Meinzer, DJ4ZC
Stephen Moraco, KZ0Q

Mechanical Drawings by:
Wilfred Gladish

Document Created by:
Stephen M Moraco, October 2003
<kz0q@amsat.org>

Last updated: 24 October 2007

Table of Contents

| | | | |
|--|-----------|--|-----------|
| 1: Introduction | 1 | A: CAN-DO! Schematics | 34 |
| Definitions..... | 1 | B: Mechanical Drawings..... | 37 |
| Widgets, Modules and IHU..... | 1 | C: CAN Message Specifications..... | 42 |
| Reading Plan | 2 | CAN Module Addressing | 43 |
| Reading Plan for the Module Developer | 2 | Standard Mode CAN Messages | 44 |
| Module Development Concepts | 3 | 0x00MM Module Configure Command | 44 |
| Interface to the CAN Do Widget..... | 4 | 0x24MM Module Status AN03..... | 44 |
| Deciding on Widget Mode | 4 | 0x28MM Module Status AN47..... | 45 |
| Implications of using an advanced mode | 4 | 0x14MM Module Census Query..... | 46 |
| Interacting with your Module over CAN..... | 5 | 0x34MM Module Census Response | 46 |
| Keeping your Widget alive during testing..... | 5 | Multiplex Mode CAN Messages | 47 |
| 2: Electrical Design | 6 | 0x00MM Module Configure Command | 47 |
| Interfacing Guidelines | 7 | 0x20MM Module Status IN08..... | 47 |
| Circuit Description..... | 7 | 0x24MM Module Status AN03..... | 48 |
| Radiation Testing Results | 8 | 0x28MM Module Status AN47..... | 49 |
| 3: Mechanical Design | 9 | 0x14MM Module Census Query..... | 49 |
| Address and Mode Selection | 11 | 0x34MM Module Census Response | 50 |
| CAN Bus and Power Cabling..... | 12 | Byte-pipe Mode CAN Messages..... | 51 |
| In-system Reprogramming..... | 13 | 0x00MM Module Configure Command | 51 |
| 4: Firmware Overview..... | 14 | 0x24MM Module Status AN03..... | 51 |
| Analog to Digital Conversion Process | 15 | 0x28MM Module Status AN47..... | 53 |
| Hardware Watchdog Recovery System | 15 | 0x04MM Byte-pipe data write to Module..... | 53 |
| Logging/Reporting of Watchdog Events..... | 16 | 0x2CMM Byte-pipe data read from Module . | 54 |
| CAN Traffic Health Indication | 16 | 0x14MM Module Census Query..... | 54 |
| Analog Sensors Common to All Modes..... | 17 | 0x34MM Module Census Response | 55 |
| 5: Standard Mode..... | 18 | Diagnostic Messages (All Modes) | 56 |
| Sample interface Schematics | 20 | 0x18MM Module Read Counters | 56 |
| 6: Multiplex Mode | 22 | 0x38MM Module Read Counters Response . | 56 |
| Rationale for two multiplex passes..... | 23 | 0x30MM Device Info Response..... | 57 |
| Multiplex Pins and Timing | 24 | 0x70MM ERAM Dump Response | 59 |
| Sample interface Schematic | 27 | 0x1CMM Module Jump to ISP Loader | |
| 7: Byte-pipe Mode | 28 | (deprecated)..... | 60 |
| "Configure" Traffic | 29 | 0x3cMM Module Jump to ISP Loader ACK | |
| Byte-pipe Read/Write Traffic..... | 29 | (deprecated)..... | 60 |
| Byte-pipe versus Module Power | 29 | D: Control Software Development..... | 61 |
| Byte-pipe power-on stabilization interval..... | 30 | Reading Plan for the Control-software | |
| Transfer Stall Recovery | 30 | Developer..... | 61 |
| Byte-pipe Pins and Timing | 31 | Control Software Concepts | 62 |
| | | CAN vs. I/O Multiplexer and CAN Errors..... | 62 |
| | | Bulk Data – to/from Modules..... | 63 |
| | | Widget recovery | 63 |
| | | IPS-centric discussion of Widget constellation | |
| | | control by Bdale, KB0G..... | 64 |
| | | E: Revision History | 65 |

List of Figures

| | |
|--|----|
| Figure 1 - CAN DO Widget (in red) Context | 1 |
| Figure 2 - Widget angled top views showing 40 pin connector orientations | 9 |
| Figure 3 - Widget side views showing 40-pin connector orientations | 10 |
| Figure 4 - Location of Address and Mode Pads..... | 11 |
| Figure 5 – Schematic: Simple 1 widget cable..... | 12 |
| Figure 6 - Author’s boot-mode selector and reset switch..... | 13 |
| Figure 7 - Standard-mode Widget I/O Overview..... | 18 |
| Figure 8 - Example 10V Logic Level Module Interface..... | 20 |
| Figure 9 - Example 5V Logic Level Module Interface | 21 |
| Figure 10 - Multiplex-mode Widget I/O Overview | 22 |
| Figure 11 - Multiplex byte write (to Module from Widget)..... | 25 |
| Figure 12 - Multiplex byte read (from Module to Widget)..... | 25 |
| Figure 13 - Multiplex eight Latches being written/read..... | 26 |
| Figure 14 - Example multiplex decoding with 1 of 8 latch-pairs shown..... | 27 |
| Figure 15 - Byte-pipe-mode Widget I/O Overview | 28 |
| Figure 16 - Coordination between sender and receiver..... | 32 |
| Figure 17 - Example Byte-pipe Handshake | 32 |

List of Tables

| | |
|--|----|
| Table 1 - Meaning of Widget Mode Settings..... | 11 |
| Table 2 - Measurements from released firmware v3.04..... | 24 |
| Table 3 - Byte-pipe Pin Assignments..... | 31 |

List of Equations

| | |
|---|----|
| Equation 1 - Analog Sensor value to Volts..... | 17 |
| Equation 2 - Widget Current Draw (iLoad)..... | 17 |
| Equation 3 - Widget Temperature in Celsius (tempC)..... | 17 |

1: Introduction

A brief review of concepts and terminology

Definitions

| | |
|----------------|--|
| Widget | CAN Interface Board |
| Module | A box of stuff on spacecraft that includes a Widget |
| Bus | CAN (Controller Area Network) |
| Mode | Standard, Multiplexed, or Byte-pipe |
| Address | Each Widget on the CAN bus must have a unique non-zero 6-bit address |

Widgets, Modules and IHU

A CAN-DO Widget serves as a Module's interface to the CAN Bus. A “master” controller for the CAN bus hereinafter referred to as the IHU (Integrated Housekeeping Unit) controls these widgets. While we speak in terms of an IHU, this should not be construed to mean any particular computer or controller, but to refer generically to the system masters.

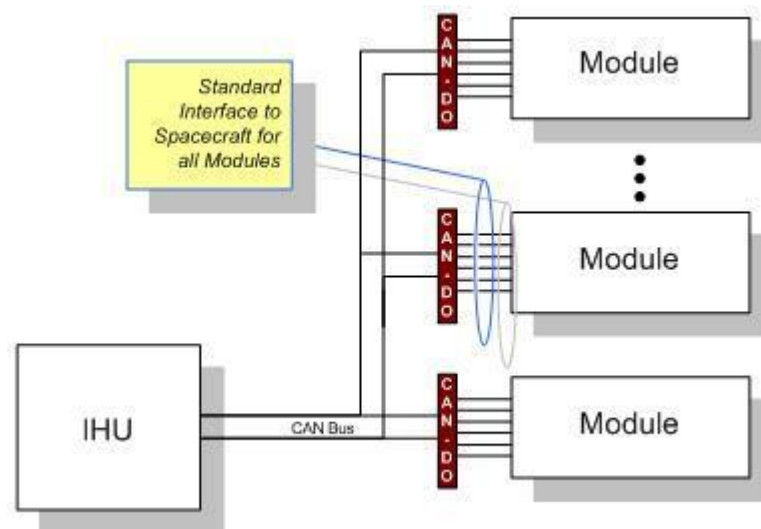


Figure 1 - CAN DO Widget (in red) Context

These widgets provide a standard module interface to the spacecraft consisting of digital outputs, digital inputs, analog inputs, switched power, current sensing, and temperature sensing. There are three possible modes for which a widget can be configured that differ in the number of digital inputs/outputs and analog inputs and

INTRODUCTION

how these are presented to the module. The three modes currently defined are Standard, Multiplexed, and Byte-pipe.

Separate chapters later in this document describe each of these operating modes in more detail. You will also find chapters on CAN-Do Widget electrical and mechanical interfacing. The remainder of this chapter, however, presents suggestions for an effective reading sequence of this information.

Reading Plan

A wealth of information about our widgets, how they are to be used, how to interface with them, how they work, etc. is provided in this document. Your time is valuable. In this section, we recommend a reading plan for you our Module (or Payload) developer for learning what you need to know and where it is found in this document.

*We have provided, in **Appendix D: Control Software Development**, similar information for those planning to develop Control Software (e.g., firmware in the IHU or another widget network controller of some sort).*

Reading Plan for the Module Developer

Welcome Module Developer to the world of Widget integration! In this section, we will help identify what you need to know to make integration with the Widget and by extension the satellite much easier than you might have experienced with past satellites. As we are identifying what you need to know, we will also be identifying where to find specific related information in this users guide and in some cases at locations on the worldwide web. Let us get started.

There are a number of tasks facing you the Module Developer having to do with use of the CAN-Do Widgets. These are:

- Identify the of number of control signals, and the number of telemetry signals needed by module
- Select Widget Mode to support the needed number of signals
- Possibly add latch hardware in support of the needed Widget Mode (Multiplex Mode only) or add pipe-mode support to your modules microprocessor firmware (Byte-pipe Mode only)
- Design control-signal and switched-power handling so that Watchdog resets of the Widget will not interfere with the function of your Module
- Acquire a CAN Interface for the PC which will be used during module development and testing
- Acquire a suitable power supply for the Module test environment
- Construct the CAN/Power cable which connects the Module to the CAN interface on the PC and to the power supply
- Acquire free software we have created for your use during development of Module that simulates full IHU functionality and provides richer diagnostic support and control than when connected to IHU/Spacecraft
- Setup the test environment, which includes Module under development, associated PC, and configuration of the Widget via jumpers, attaching the Widget to the Module, cabling everything up, etc.
- Develop, test and refine the Module

INTRODUCTION

- Document the control and telemetry signals for this module (where they are in the CAN messages for this Module)
- Coordinate with the IHU development team the adding of control and reporting for the new Module

Please review the following sections of this User's Guide to obtain information needed to accomplish these tasks:

1. Read the section entitled “**Module Development Concepts**” to obtain a conceptual overview of areas in support of your efforts.
2. Read the section entitled “**Interfacing Guidelines**” within chapter “**2: Electrical Design**” to learn of the electrical requirements for interface with the Widget's 40-pin connector.
3. Skim the three mode specific chapters to learn of why each mode exists and to review example-interface schematics in order to obtain an initial idea of how you will be interfacing the Widget to your module.
4. Once you have determined the Widget Mode most suitable to your application then review in detail the mode specific chapter for your chosen mode.
5. Refer to Appendix “**A: CAN-DO! Schematics**” when developing your Module Schematics. Remember that you will be adding additional latch hardware if you have determined that the Multiplex Mode is appropriate for your design.
6. Review chapter “**3: Mechanical Design**” as well as engineering diagrams [E05-40, E05-45, and E05-46] found at the AMSAT-NA EaglePedia (<http://www.amsat.org/amsat-new/eagle/EaglePedia>) when you are planning your initial PCB layout and as you are finalizing your layout.
7. Review in detail the Mode Specific section of the Appendix “**C: CAN Message Specifications**” and then create draft documentation of which CAN packet bits mean what relative to your module implementation. This will help guide the testing of your module and the creation of the IHU software interacting with your Module.
8. Review the section entitled “**Address and Mode Selection**” of chapter “**3: Mechanical Design**” to learn how to jumper your widget for the selected mode.
9. Review the section entitled “**CAN Bus and Power Cabling**” of chapter “**3: Mechanical Design**” to learn the details of constructing the CAN/Power cable you will need for testing of your Module.

As you are reading sections of this User's Guide, you will see there is much more information here than this reading plan suggests. Not all of this information may be useful to your task but you may find a read through this additional material may help you understand more of how and why the Widget is what it is today.

Module Development Concepts

This section introduces you the **Module developer** to the electrical interface of the Widget and the bus shapes it supports. It helps you decide which bus shape (represented as Widget operating Mode) best supports the needs of your Module. It introduces you to a few “special needs” of the more advanced Widget operating modes that you will need to take into account in your Module design in order to use these modes. Lastly it introduces the free software which is already written for you which you can immediately use to interact with your new Module.

Interface to the CAN Do Widget

The Module's interface to the widget is a 40-pin connector, which provides power, analog telemetry and bidirectional digital signals (control and telemetry) between the Module and the IHU. This is a parallel bus. The latched control values provided to the module are refreshed at a fixed rate (20mSec to less than 3Sec.) by the Satellite controller (IHU, or ground test software running on a PC). During each of these refresh updates the telemetry from this module is reported back to the controller at in response to each refresh. As a module developer, you get to choose the bus structure that the Widget will present to your Module electronics (from three different bus structures). This choice is made by shorting pads on the Widget during Module construction and cannot be changed after the Module is closed up and the satellite is launched. These three choices offer different numbers of control (digital) output signals to the module from the controller and a different number of telemetry inputs. Once you have determined the scope of control and status reporting your module needs then you can choose the appropriate bus width for the application and configure your Widget appropriately.

Deciding on Widget Mode

The decision of Widget operating mode to use is critical but not difficult. First, the majority of Modules on a given spacecraft will likely use Standard Mode. There are special types of communication, however, which may require you to choose one of the advanced Widget modes. In general the decision reasoning is:

- a. If Module contains a microprocessor that needs to exchange bulk data (download of pictures, upload of tables, etc.) with the IHU or if the microprocessor within the Module needs to have code loaded then the use of Byte-pipe Mode is required.
- b. If you need 13 to 63 digital outputs or 9 to 64 digital inputs than you need to use Multiplex Mode.
- c. If you do not require the capabilities offered by the above two modes than you want to use Standard Mode.

A final qualifying thought: The widget does not support the use of more than 63 outputs or 64 inputs.

In summary then think of the mode choice as use Standard Mode unless the guidelines above indicate that you need to use an advanced mode. The advanced modes require additional supporting work by the Module development team and therefore should only be used when needed. Let's look at the additional work.

Implications of using an advanced mode

The widget provides a core set of capabilities, which the standard Mode offers. In order to support the two advanced modes the module implementation needs to accept adding a part of the functionality to complete the mode behavior. In the case of the Multiplex mode, we require the addition of a small bit of hardware to the Module. This additional hardware adds latches with latch address decoding to complete the latched multiplexer behavior. However, in the case of the Byte-pipe mode instead of needing additional hardware we impose two Module implementation constraints, which the Module designer can meet through some combination of additional Module (not Widget) firmware and possibly additional logic:

- (1) A performance constraint on the offload of pipe-data from the Widget (since the Widget has precious little memory in which to hold data on its way to the Module from the IHU)

INTRODUCTION

- (2) A specific bulk data transport protocol constraint (All Byte-pipe Modules implement the same transport protocol thereby allowing us to implement one transport protocol in the IHU for use by all manner of microprocessor-controlled Modules. Additionally this protocol works well given the constraint of the limited-buffer-space within the Widget.)

Interacting with your Module over CAN

You have the Widget and you've decided on operating Mode for the widget and now you need to begin exchanging CAN packets with the widget. In order to do this you will need a personal computer running Linux or Windows and you will need a CAN interface so programs can be run on the computer which can send and receive CAN packets via the interface to your new Module. We recommend the use of one of two CAN interfaces (as only they are supported by our test software):

- (1) Lawicel CAN232 serial interface (lower performance, lower cost)
<http://www.can232.com/>
- (2) Peak Systems PCAN USB device (higher performance, higher cost)
http://www.peak-system.com/db/gb/pcanusb_gb.html

After procuring one of these supported CAN interfaces you can now use one of two available software programs for interacting with CAN depending upon which PC operating system you are running:

- PC running Linux
 - User Housekeeping Unit (UHU) software
(designed for interaction with only one Widget on a cable)
- PC running Windows
 - User Housekeeping Unit (UHU) software
 - CAN Do Network Controller software (CDNC).
(designed to interact with one Widget on a cable to an entire satellites worth of Widgets on the cable)

These programs support both the Serial and the USB CAN controllers. Links to downloading either of these programs can be found at the CAN Do website at our the reference page (<http://can-do.moraco.info/reference>)

Keeping your Widget alive during testing

The Widget contains a hardware watchdog, which will reboot (power-on reset), the Widget if no communication is heard from the control software or IHU at least once in a 3.12 second interval. Both UHU and CDNC have modes which allow you to turn on the configure heartbeat at a fixed rate. Both tools allow you to adjust this rate. If you are using the CAN232 interface device, you may not be able to run at the full 50 Hz rate that the IHU runs. In addition, both UHU and CDNC allow you to adjust control values to be sent to your Module while this heartbeat is active so you do not need to stop the heartbeat in order to change control values being sent. This is mentioned in this section as we have experienced developers who were trying to understand why they could not get a specific test to run without failure over a period longer than 3.12 seconds. If you keep the Widget Watchdog from firing by sending configures (at least one every 3 seconds) then you will be able to run tests without interruption.

This completes the Module Developer introduction. Please continue with the next item in the reading plan you are following.

2: Electrical Design

Overview, Interfacing Guidelines and Schematic Walkthrough

The core component on the "widget board" is an Atmel T89C51CC01 microcontroller. This part is a derivative of the venerable Intel 8051 architecture, including a number of digital input and output lines, an integrated analog to digital converter, a CAN interface, and onboard flash program memory and data RAM.

Jumper locations on the board allow for six bits of address selection allowing up to 64 modules, and two bits of mode selection. Currently, we implement three of four possible modes... normal mode, a "multiplexed" mode supporting expansion of the digital control and telemetry line count, and a byte pipe mode for communication with a processor in the module's electronics.

In normal mode, the features provided by the widget board include a power switch, current sensor, temperature sensor, 12 digital output lines, eight digital input lines, and five user-defined analog sensor channels.

In multiplexed mode, support for up to eight banks of external multiplexers replaces the digital output and input lines of normal mode. The twelve output lines now function as an eight-bit multiplexed output data bus, a three-bit multiplex select, and a one-bit strobe. The eight input lines convert to an eight-bit multiplexed input data bus using the same select and strobe lines as the outputs. Thus, in this mode, the widget board provides a power switch, current sensor, temperature sensor, up to 63 lines of digital output and 64 lines of digital input, and five user-defined analog sensor channels.

In byte-pipe mode, the widget implements distinct eight-bit input and output busses with simple handshaking. The feature set in this mode includes the power switch, current sensor, temperature sensor, eight-bit input and output busses each with strobe/ACK handshake lines, two independent output lines, and three user-defined analog sensor channels.

We have discussed ideas for other modes we might support, including various flavors of serial byte pipe. There are also proposals for reusing this hardware with fully custom firmware for some special sensor applications. The team has made no commitments to support any of these other applications, however.

Interfacing Guidelines

Module designers need to adhere to a few simple guidelines when designing the electrical interface of their module to the CAN-DO! Widget. These guidelines are:

- 1) The CAN-Do! inputs are 5-volt level signals.
 - a) 3.3V logic signals from the user module to CAN-Do! must be buffered to 5V levels.
 - b) Signal levels greater than 5V must be scaled to 5V before applying them to CAN-Do!
- 2) The CAN-Do! output signals are configured as diode-isolated pull-downs with 1K series resistance.
 - a) The user module must provide low-current pull-ups to a +5V source. The source may be less than +5V, but not greater than +5V.
 - b) 10K ohms is a suggested *minimum* pull-up resistance. A higher value of pull-up resistance is better, if the user module can accept the weaker current and greater delay.
 - c) The user module must accept a logic LOW to be one [(Schottky diode drop + (current-current through 1K ohms)] above 0V.

For example, if the pull-up is 10K ohms to 5V, then a logic LOW = $[(0.3V) + (1K)*(5.0-0.3)/(10K+1K)] = 0.73V$.

If the pull-up is 47K ohms to 5V, then a logic LOW = $[(0.3V) + (1K)*(5.0-0.3)/(47K+1K)] = 0.40V$.

- 3) CAN-Do! provides a +5V power source. This is a low-current source, and should only be used for buffering the CAN-Do! signals and providing isolation between the CAN-Do! widget and the user module.

Possible interface circuits are provided as example schematics: **Figure 8 - Example 10V Logic Level Module Interface**, **Figure 9 - Example 5V Logic Level Module Interface**, and **Figure 14 - Example multiplex decoding with 1 of 8 latch-pairs shown** which are found within the mode-specific chapters. These schematics attempt to show adherence to these interfacing guidelines.

Circuit Description

Please refer to the schematics found in Appendix A or at our project website for this discussion.

A 15-pin male D-sub connector (P1) interfaces to the spacecraft, brings in nominal 14V power, provides CAN connections, and passes through up to five user-definable signal lines that connect directly to the associated module. Redundant CAN connections enable the spacecraft wiring harness to "loop through" the module without splicing wires. An "EB" line passes through to the user module. This allows tapping the IHU engineering beacon data stream if required by the module.

This "EB" pin is just a placeholder for those modules that require it. The wiring harness would not normally provide the "EB" unless the user module required it. If a module does not need the "EB" then this pin can be used as a 6th "user defined" signal line.

ELECTRICAL DESIGN

A 40-pin connector (J5), organized as two rows of 20 pins all on 2.00 mm centers, attaches to the module and brings all available signals to the module.

Incoming power is fused, filtered and applied to a power switch (FET Q2 and associated electronics) and to a switching regulator (U7) which provides the required +5V to run the CAN module.

Analog circuitry includes a temperature sensor (U12) and a means to monitor the current consumption of the attached module (U8). Remaining analog circuitry conditions the ADC inputs with filtering and clamping. Analog signals are then applied to the eight ADC inputs of the microcontroller (U5).

CAN signals are filtered by U3 and applied to U4, which converts from the CAN physical-layer signaling scheme to standard CMOS 5V logic levels. U4 has its own power supply filtering to minimize noise effects from the CAN bus. The CMOS signals interface directly to U5.

U5 includes clock circuitry, operating at 8 MHz set by crystal Y1. U6 provides a regulated analog voltage reference of 2.5V independent of the accuracy of the 5V supply.

NOTE: with a V_{AREF} of 2.5V, any $AN[0-7]$ value $\geq 2.5V$ yields a maximum conversion value of 0x3ff. Any value $AN[0-7] \leq V_{AGND}$ yields a 0x000 conversion value. The voltage at $AN[0-7]$ must not exceed $V_{AREF} + 0.2V$ or 2.7 Volts. The conversion is straight-line linear with a resulting step size of $2.5V/1024$ steps (2.44mV per step).

Shift register U2 allows the microcontroller to read the jumpers at reset or on command. The jumpers set the widget address as well as its operating mode.

Schottky diodes and current limiting resistors provide isolation for the digital outputs of U5. The user module must provide pull-ups to not more than +5V, and the pull-up value should be at least 10K ohms.

Radiation Testing Results

We were pleasantly surprised by how well the Atmel microcontroller survived our radiation testing. By arrangement with the University of Virginia Medical Center, prototype CAN widgets were exposed to a calibrated radioactive source in controlled dosages. We evaluated the modules power consumption and functionality after the exposure. We repeated this radiate/evaluate cycle until failure, which occurred at some 60k rads.

In addition to the confidence this testing provided, we directly applied some of its results to the circuit design. For example, we decided to use the clock oscillator on the microcontroller, which saves tens of milliwatts of continuous power consumption. This may result in a power savings of a watt or so in a moderately complex spacecraft.

To enhance reliability in the radiation intense GTO orbit environment, AMSAT will continue our traditional practice of applying additional shielding material to critical ICs. This shielding will likely consist of 1mm of Lead affixed to the relevant components.

3: Mechanical Design

Design of the PCB Layout and Mechanical Module interfacing

Detailed dimensioned mechanical drawings are found in Appendix B

The mechanical objectives of CAN-DO are simple to state. Keep the board as small as possible while maintaining a form factor that fits into any size box available to the module developers, and provision for mounting the board adequately to withstand vibrations anticipated during launch.

The board height of 24mm fits into the minimum box thickness of one inch. The 67 parts on the board (not counting radiation shields) range in size from 0603 resistors to the 15 pin Sub-D connector. The width of 74mm is what is necessary to contain all the parts.

The 15 pin Sub-D connector established the space needed between the PCB and the inside edge of the box it is mounted to, about 1/4 inch. In order to achieve efficient use of volumetric space all high-profile parts are on the side of the PCB with the 15 pin connector and only low profile parts are placed on the back side of the PCB. The tallest part on the back of the PCB stands out about 3.5mm with the exception of the 40-pin connector that plugs into the module PCB.

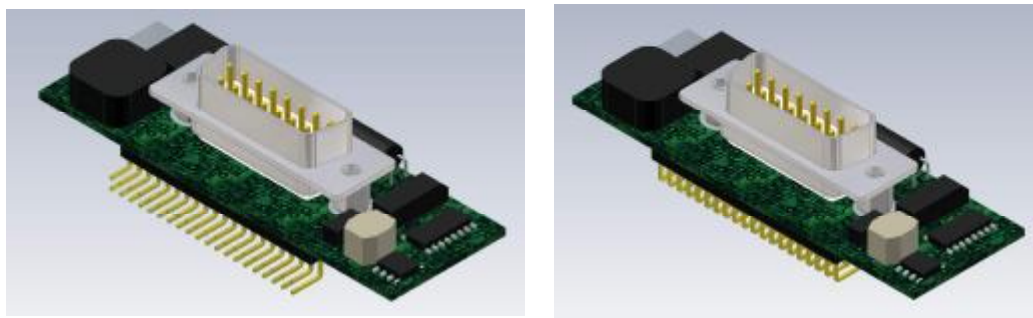


Figure 2 - Widget angled top views showing 40 pin connector orientations

In some cases, a module developer may wish to extend the PCB all the way to the connector side of the box. If the box height is sufficient, this is accomplished by inverting the orientation of the 40-pin connector so that the right angle pins point down rather than up. This connector is located at the bottom edge of the PCB to accommodate this choice and centered left-to-right along this edge. Error! Reference source not found. and Error! Reference source not found. (next page) show these two connector orientations.

The 15 pin Sub-D connector is located in the geometric center of the PCB. It is mounted to the PCB by stand-offs so that when the connector is secured to the side of the box, the PCB is also securely held.

MECHANICAL DESIGN

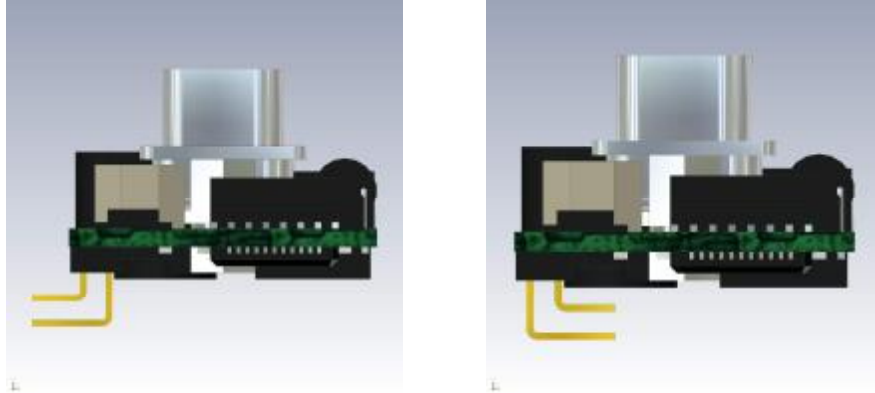


Figure 3 - Widget side views showing 40-pin connector orientations

Wherever a part requiring a radiation shield (IC's and FET transistors) is located on the PCB, a similar part is located on the opposite side of the PCB. This way, a radiation shield attached to the top of one part also shields the bottom of another part thereby reducing the total number of shields needed.

The FET module power switch is mounted on the side of the PCB facing the box edge to maximize heat radiation coupling to the side of the box rather than back into the module contained in the box. Similarly, the thermistor is on the backside of the PCB facing the application module.

The above requirements obviously made component placement challenging, but an even bigger challenge was connecting everything properly. Some traces are wide to allow for significant currents. Other traces are very narrow. The narrowest traces on the PCB are 0.006 inches wide and the smallest spacing between traces is 0.006 inches. This is a four-layer board.

Most components are SMD and these units are assembled by hand.

Address and Mode Selection

Address and Mode selection for a given Widget are configured by shorting, or leaving open, pads. These pads are located on the side of the Widget opposite of the D15P connector and at the edge opposite of the 40pin Connector.

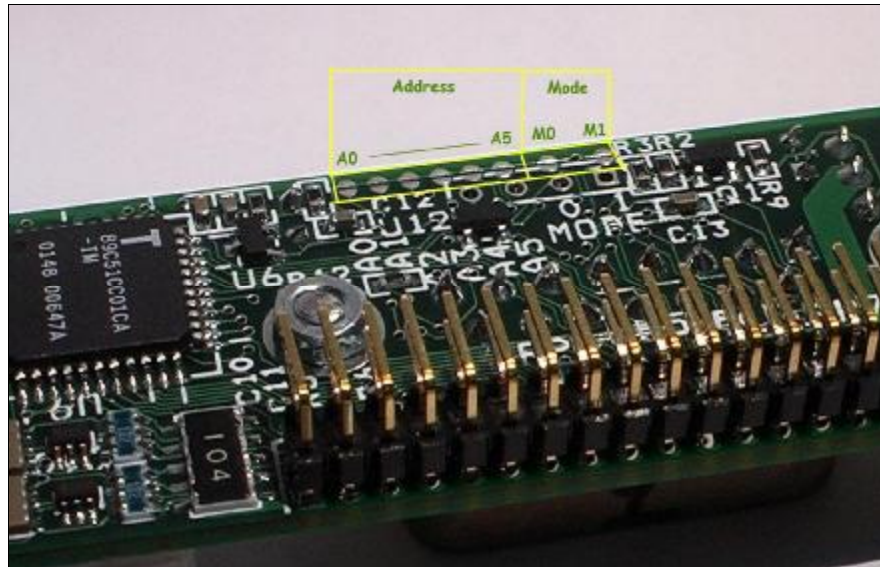


Figure 4 - Location of Address and Mode Pads

There are six pads for address (A0-A5) yielding address from 0x01-0x3F (NOTE: 0x00 must remain unused) and two pads for mode (M0, M1). Error! Reference source not found. (above) shows the location of the pads on an actual Widget. Table 1 provides the definition of the Mode bit values.

| Mode Bit (M1) | Mode Bit (M0) | Meaning |
|---------------|---------------|-------------------------|
| Shorted – 0 | Shorted – 0 | Reserved (0x00) |
| Shorted – 0 | Open – 1 | Byte-pipe Mode (0x01) |
| Open – 1 | Shorted – 0 | Multiplexed Mode (0x02) |
| Open – 1 | Open – 1 | Standard Mode (0x03) |

Table 1 - Meaning of Widget Mode Settings

Address and Mode bits default to one (open pad). Shorting a pad sets the bit to zero. These pads are located on the schematic as JP1 - JP8 on Sheet 1 of 2.

Widget address assignments are made carefully as addresses affect module communication priority on the CAN bus. Widget address assignments are unique to each spacecraft. The Spacecraft's Project Manager therefore determines widget address assignments. See Appendix C "CAN Module Addressing" for more information on address priority.

CAN Bus and Power Cabling

Creating a 1-widget cable for use on your test bench

CAN devices are wired to the bus in parallel. A simple 120-Ohm resistor should be placed at each of the two extreme ends of the bus, between CAN-HI and CAN-LOW, for termination. The D15P connector provides for two CAN-HI and two CAN-LOW connections to simplify the creation of the wiring harness.

The typical wiring harness for a single CAN232 controller and a single widget is very simple. It consists of the following parts:

- ✚ 1 – DB15S
- ✚ 1 – DB9S
- ✚ A length of cable to be used for power (2 conductor)
- ✚ A length of cable to be used for the CAN communication (2 conductor)
- ✚ 2 – 120 Ohm resistors
- ✚ 1 – Power connector socket

Power routes to both the Widget and to the CAN232 Bus adapter. Error! Reference source not found. shows a schematic of this configuration.

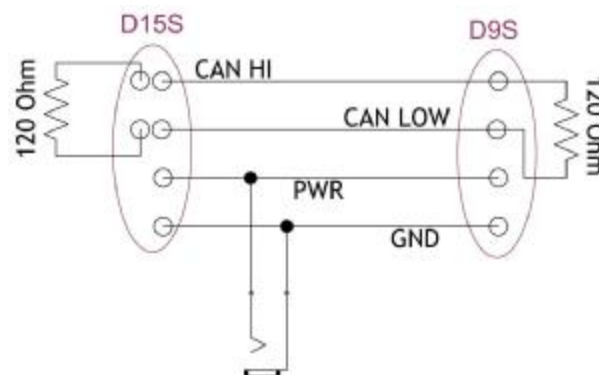


Figure 5 – Schematic: Simple 1 widget cable

Note1: for D9S pin numbers when using Lawicel CAN232 device see: <http://www.can232.com/> For the D15S pin numbers see connector P1 on page 2 of the CAN-DO schematic in Appendix A.

Note2: The CAN232 device requires that PWR be 8-15 VDC.

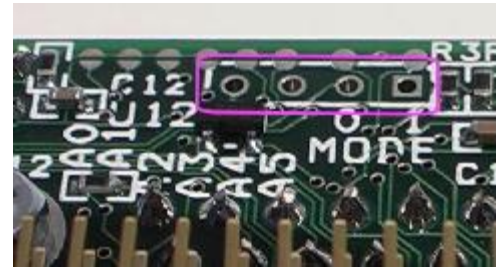
The author uses a more complicated harness which connects 3 widgets, one test jig (IHU simulator), one CAN bus adapter for monitoring and another CAN bus adapter (the Lawicel CAN232). When using a cable built for more than one widget, shorting plugs were created out of D15P's which allowed attaching a shorting plug at each D15S where a widget was not connected so the cable could be used as a 1, 2 or 3-widget cable

In-system Reprogramming

A module developer should not need this capability. This facility is presented here for completeness.

The microcontroller onboard the CAN-DO widget provides a power-on reset capability. However, we will not use this in flight so there is no reset switch on the Widget board. Connection pads for a reset switch are present, though to support In-system Re-programming, that is, reloading of the onboard firmware. The Module developer should not need to use this power-on-reset capability. Simply powering down and re-applying power will be sufficient to restart the widget. The rest of this section describes the use of these reset connection pads.

Four lines are brought out to a set of plated-thru holes (P2 ISP) in a standard connector configuration, so simple external reset hardware may be temporarily attached. [See picture to right, purple outline shows location of P2 pins 1-4. Pin 1, the square pad, is to the right.]



There are two reset types: (1) restart and jump back into the firmware or (2) restart jumping into the built-in CAN boot loader in order to accept new firmware. The author created a simple slide switch and push-button combination device (Error! Reference source not found..) which allows switching between the two reset forms and then causing a reset by pressing the push-button. The push button shorts pins one (VCC) and two (RESET) when pressed. The slide switch connects a 100 ohm resistor between pins three (PSEN) and four (GND). If the resistor is in-circuit at reset pushbutton release, the device jumps into the boot-loader. If the resistor is not in circuit at release the device jumps into the current firmware. These parts are mounted on an 8-pin DIP socket with one row of pins cut off.



Figure 6 - Author's boot-mode selector and reset switch

4: Firmware Overview

Behaviors Common to all Modes

The hardware in the Widget was chosen to provide a basic set of capabilities. In addition to providing control of this hardware our firmware implements three bus shapes we call Standard, Multiplexed and Byte-pipe. Because these shapes come from studying satellites, which have flown, they fulfill most of our wiring harness needs. The next three chapters describe how the firmware realizes each of these bus shapes in detail. Before we get to the detail, though, this chapter introduces the basic behaviors provided by the firmware, which are common to all bus-shapes. Each of the bus-shapes may then slightly alter some of this behavior.

Taking a more general system view for the purposes of this discussion we have the IHU to which is wired a number of payloads (Modules) all connected via the CAN bus. The IHU will write control values to all of these Modules and record the telemetry from each of these Modules (both analog channels and digital inputs) 50 times a second (or every 20 mSec.). Each Module contains a Widget which is its interface to the CAN Bus. The IHU is effectively sending a configure packet to each Widget on the bus, to all of the Widgets, within this 20 mSec period. Each configure packet is addressed to one specific Widget. The number of configure packets sent is equal to the number of Widgets on the CAN bus. The Widget intended to receive the configure packet is then expected to digitize all built-in analog channels and read all digital inputs returning both sets of values as one or more answer packets. The CAN bus is fast enough to transfer all “configure” packets to the Widgets and to receive all “response” packets from the Widgets in each 20mSec period.

The design intent of this system is that the digital output lines from the Widget to the Module are held at a constant value (latched) and only change when a new configure packet has arrived and is processed. The expectation also is that the Module electronics hold (latch) the digital input values making them available for reading by the Widget at any time. In general then, the new output values are latched and presented to the Module when the configure packet arrives, the analog channels are digitized, then the digital inputs from the Module are read and both the analog and digital values are sent in response packets back to the IHU.

As a design clarification: We see that the configure packets arrive at the Widget 50 times a second. Therefore, if one were to send alternating values in one of the output bits in the configure packets the highest rate of change of this digital signal we can attain is 25 Hz. A 25Hz data rate offers very low information content so we should consider these digital outputs as a parallel bus. We should not impose any serial protocols at this frequency as this dramatically reduces the overall performance of our system and imposes a need for software implementation of the serial protocol in the IHU code base, which undesirable as it increases the risk of in-flight failure due to the increased complexity of the code. From a simple data transfer perspective a 16 bit word is transmitted at this rate at ~1.25 16-bit words per second serially, or 50 16-bit words per second when in parallel. Now we see from a system perspective that parallel transfers require less software in the IHU and we have much more immediate control of our device using them. Therefore, the moral of this design

clarification is “let’s not use serial protocols (e.g., I²C, SPI) over this CAN bus.” If you still find yourself wanting to use devices that require SPI or I²C then plan on also including devices that convert parallel configure data arriving from the IHU to the serial protocol you need.

In addition to the core capability, the firmware implements additional mechanisms, which are worth understanding in slightly more detail. These are the analog to digital conversion process, the hardware watchdog-based recovery and reporting system, and the CAN traffic health-tracking system. Satellite control-software designers as well as the Module hardware designers must take into account these additional mechanisms during design and implementation.

Analog to Digital Conversion Process

There are eight analog channels provided by the T89C51CC01 microcontroller. The firmware in response to a configure packet will digitize all of the channels and report these values in response packets. The firmware digitizes all eight channels in sequence. It makes 16 digitizing passes over all eight channels accumulating a sum for each channel. At the end of the 16th pass of eight channels, it then divides each sum by 16, which produces the result reported in the answer packets. This process is not allowed to be interrupted for any reason. Additionally the microcontroller is placed into a quiet mode (less onboard activity) during this digitizing effort in order to reduce system noise, which could affect the digitized result. The process to convert all eight channels 16 times, divide each channel sum to get the result takes roughly 6.2 mSec. This means that there will be at least 6.2 mSec delay between the Widget receiving a configure packet and the Widget starting to send the answer packets. This means that an IHU will be able to send a number of configure packets before any of the answer packets start arriving at the IHU.

Hardware Watchdog Recovery System

The CAN-DO widget is part of the wiring harness. Because of this position in the spacecraft, it must be very reliable. To make a microprocessor-based system more reliable we implement a few behaviors intended to rescue a system experiencing unexpected conditions. One of the facilities of the ATMEL T89c51cc01 microcontroller is a hardware implementation of a watchdog timer. In normal operation, the firmware reports to this watchdog that it is running well. Should the firmware fail to report-in, the timer expires and the microcontroller is then reset. The intent of this system is to restart the controller when execution of the code gets the controller into some situation wherein it is not running well enough to perform its intended function. There is no software means to disable this hardware Watchdog which makes it ideal for use in this fashion.

Onboard the Widget, the watchdog (WD) timer is set to expire in roughly 3.12 Seconds (Which is the maximum duration of the watchdog timer with our Widget operating at the 8 MHz clock frequency.) In the spacecraft context we expect the Widget to be receiving configure packets every 20 mSec and we expect that the Widget is sending one or more answer packets every time it receives a configure packet. In order to keep our watchdog from expiring we simply report to it that the firmware is running well each time the Widget sends a response to the IHU. Then if for some reason the Widget stops sending responses or stops receiving configure requests to which to respond, the watchdog will reset the Widget after 3.12 seconds of this inactivity according to the assumption that it might be the Widget, which is no longer functioning well.

Providing this watchdog does come at a small expense to our Module developers. When the watchdog expires, the microcontroller resets itself. Upon reset, all I/O pins are driven high. This means that our Module circuitry must be designed so that a high value on the control lines is the non-asserted state. In order to keep these reset event from clobbering a Module we implement a detection mechanism in the Widget which can tell

FIRMWARE OVERVIEW

if a watchdog reset has occurred and will restore the last known-good configure values to the output pins as quickly as possible after restart. With sufficient capacitance on the switched power line, an attached Module should be able to get through these events with little or no effect on its behavior.

In order to be tolerant of infrequent bit errors in our RAM the firmware maintains more than one copy of the last known good configure value in RAM. When a watchdog reset occurs, and the microcontroller is restarting, the firmware compares these copies and if they are identical will restore these last known good values to the output (Module control) pins. In the case were these are not identical the Widget will simply not adjust the control outputs (which are all still high due to the reset) until the next configure packet is received which should happen in 20 mSec or less.

Logging/Reporting of Watchdog Events

The Widget firmware implements a Watchdog (WD) event log written to in ERAM (an extended, lesser used area of memory within the ATMEL processor). The newly extended version of the “0x18MM Module Read Counters” diagnostic message provides for retrieval of this WD event log by the IHU if desired (or by ground test software such as CDNC, or UHU during development of the module). This log tracks two kinds of WD events: (1) “no configure traffic heard, so resetting”, and (2) “firmware was locked in an infinite loop” when the WD expired.

This WD event log simply contains a count of the “nothing heard” expirations since there is no additional useful data to be logged. In the case of the infinite loops, however, the firmware logs which loop the firmware was in at the time of expiration. The intent of this logging is to allow diagnosis of which subsystem is failing. (This of course assumes that we have enough of the system working that we can download the WD log for analysis ;-)

The implementation of the Widget firmware uses as few infinite loops as possible. However, some of these are useful when dealing with correctly functioning hardware (less code, therefore faster response times, etc.) The firmware upon entering one of these infinite loops (e.g., waiting on a hardware event that should succeed) makes note that it is entering the loop and then notes when it successfully leaves the loop. Then should a WD event occur the restart mechanism looks for indication of having been in one of these loops and records the appropriate form of event log entry.

In summary then, Watchdog events can be indicative of transient or progressive failures of our Widget and there is diagnostic value especially during ground based testing and maybe even when we are in space to monitoring the WD event log within our Widgets.

CAN Traffic Health Indication

Each of these Widgets sits on the CAN bus and each experiences errors differently for a number of reasons. The CAN controller within the ATMEL microcontroller tracks a set of transient errors or warnings during packet transmission and reception. The Widget firmware extends this tracking by adding counters of these events, which are updated as a packet is sent or as a packet is received. The firmware implements the “0x18MM Module Read Counters” diagnostic request and “0x38MM Module Read Counters Response” messages enabling retrieval of these counters by the IHU if desired (or by ground test software such as CDNC, or UHU during development of the module). The request message can also be used to clear (reset to zero) these counters.

Analog Sensors Common to All Modes

Three of the eight analog sensor lines (AIN5 thru AIN7) have a dedicated purpose on the Widget. The remaining channels (AIN0 thru AIN4) are available for tasking by Module Designer. The three dedicated sensors are:

- AIN7: the Widget switched-power current sensor (an LT1787HVHS8 device)
- AIN6: the Widget temperature sensor (an LM60CIM device)
Linear 6.25mV / DegrC (174mV to 1,205mV : -40C to +125C)
- AIN5: the Widget switched-power current measuring circuit bias

Note: The value of AIN5 combined with the value of AIN7 provide the single current sense value for the Widget.

The formulas used by the CDNC/UHU test software are:

$$sensorValueInVolts = rawSensorValue \times \frac{2.5V}{1024}$$

Equation 1 - Analog Sensor value to Volts

$$iLoad = \frac{ABS(AIN7volts - AIN5volts)}{400\mu A / V \times 287K \times 0.02Ohms}$$

$$iLoad = \frac{ABS(AIN7volts - AIN5volts)}{0.0004 \times 287,000 \times 0.02}$$

Equation 2 - Widget Current Draw (iLoad)

$$tempC = ((AIN6volts - valueAtMinus40C) / valuePerDegrC) - 40$$

$$tempC = ((AIN6volts - 0.174) / 0.00625) - 40$$

Equation 3 - Widget Temperature in Celsius (tempC)

The next chapters present each of the three bus-modes in detail. Full CAN packet detail is found in *Appendix C: CAN Message Specifications*.

5: Standard Mode

Behavioral description and pin-outs

In Standard Mode, the Module appears to an IHU as having 12 Digital Output signals (IHU to Module), 8 Digital Input signals (Module to IHU) and 5 Analog Inputs (Module to IHU) plus Module Power Control, current sensing and temperature sensing. **Figure 7** shows this configuration:

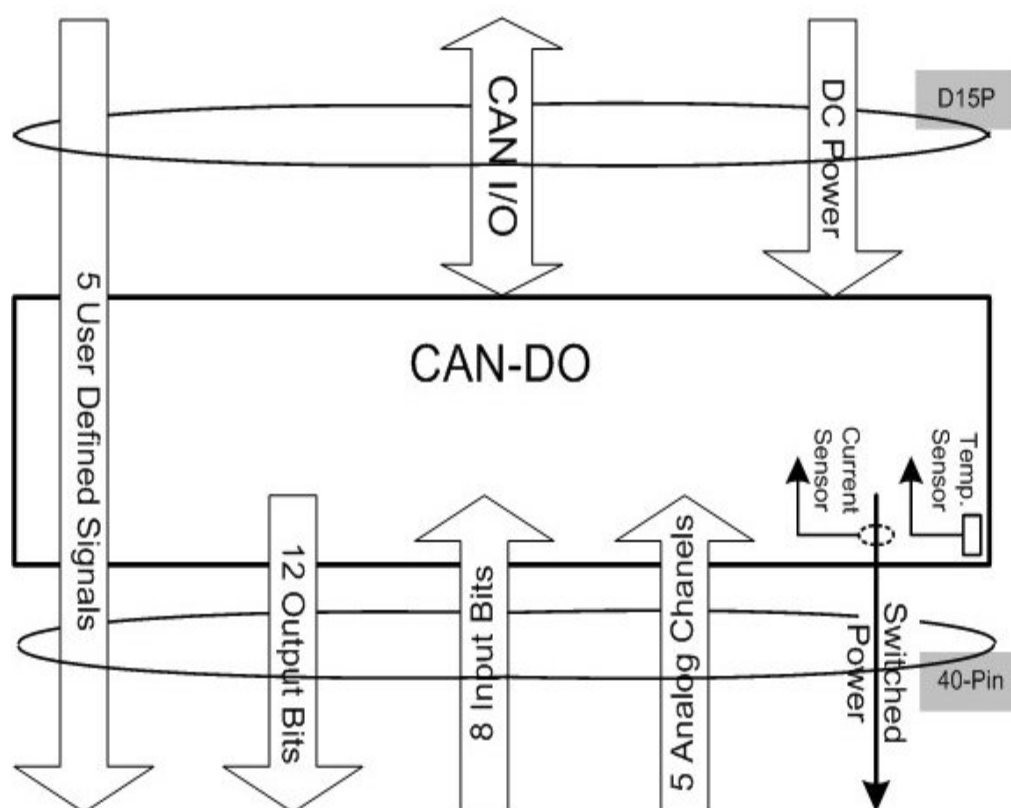


Figure 7 - Standard-mode Widget I/O Overview

In this configuration, an IHU can write all 12 output bits and power control, read the eight input bits and the five Module defined and three dedicated analog input channels (10-bit precision). It should be able to do this with all the Modules on the CAN bus every 20 milliseconds.

An IHU accomplishes this by placing the 12 output bit values and the power control value into a single 2-byte CAN packet, addressing it to the Module, marking it as a "configure" stream and sending the packet over the CAN bus.

OPERATING MODES - STANDARD

Upon receipt of this "configure" packet the Widget adjusts Module power, if requested, and then writes the 12 output values to the output lines. It reads the eight input lines, then digitizes all eight analog channels (Temperature, Current, Current Bias, and 5 Module defined) and sends two 8-byte packets back to an IHU which contain (1) the first four Analog digitization results plus 4 of the digital input values, and (2) the last four Analog results plus the remaining four digital input values. *For precise packet descriptions, see: "Appendix C: CAN Message Specifications" later in this document.*

Sample interface Schematics

Two schematics are provided in this section: **Figure 8 - Example 10V Logic Level Module Interface** and **Figure 9 - Example 5V Logic Level Module Interface**.

These are not circuit recommendations; they are untested. They are only intended to provide clarity for the requirements of this interface:

- The user module must not load the CAN-Do! widget if the module is powered off.
- The user module must provide pull-ups to +5V on all CAN-Do! outputs that are used by the user module.
- The CAN-Do! widget 5V power is only for driving a few milliamperes at most.
- If the user module is powered off, all drive signals must be either tri-stated or driven LOW.

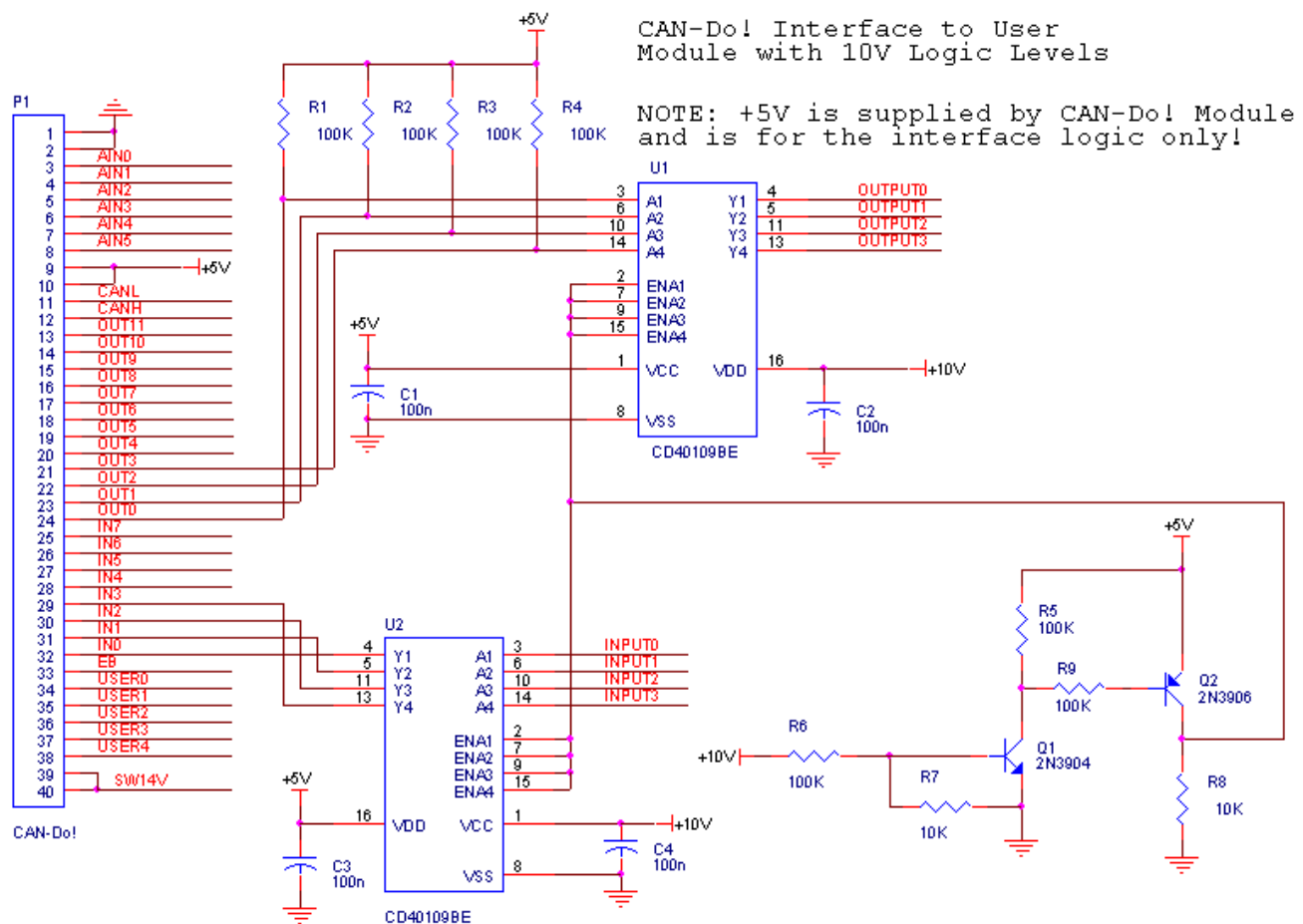
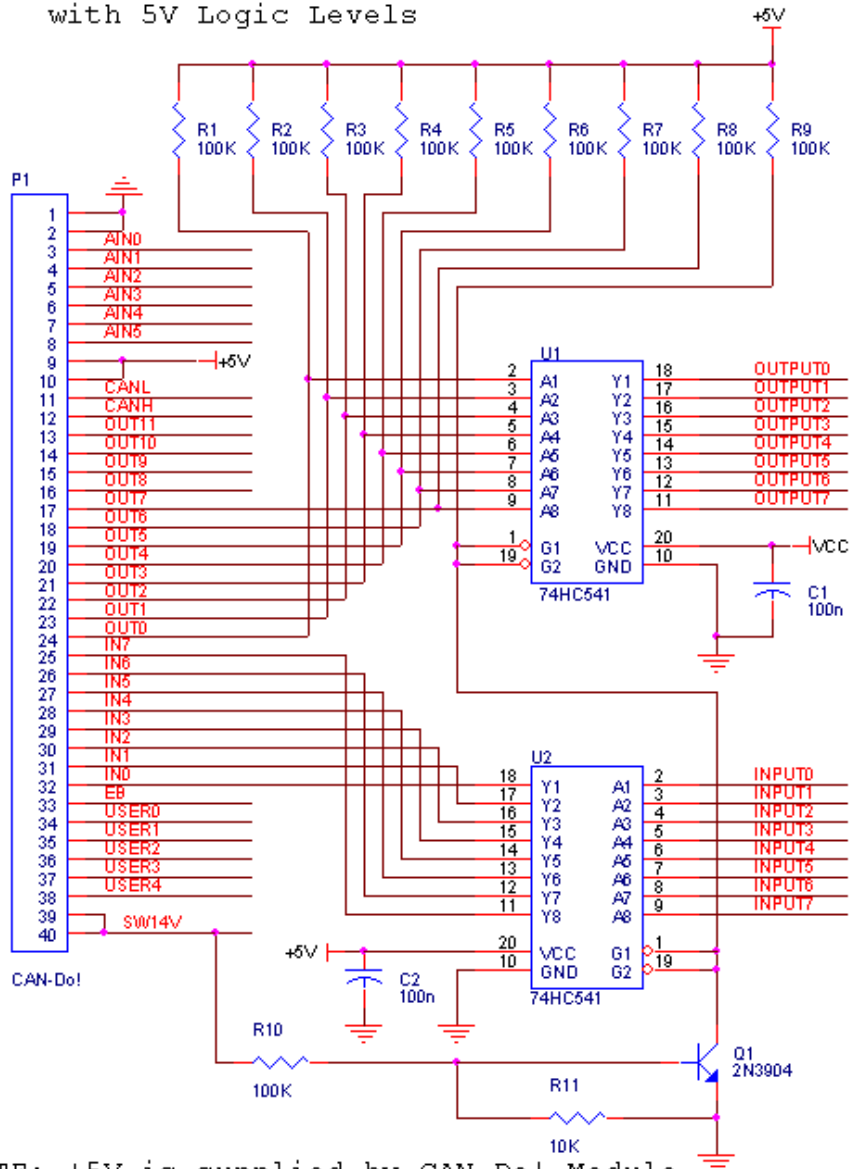


Figure 8 - Example 10V Logic Level Module Interface

OPERATING MODES - STANDARD

**CAN-Do! Interface to User Module
with 5V Logic Levels**



NOTE: +5V is supplied by CAN-Do! Module
and is for the interface logic only!

Figure 9 - Example 5V Logic Level Module Interface

6: Multiplex Mode

Behavioral description and pin-outs

The CAN Widget, when configured for Multiplex mode, requires external latches and 3-to-8 latch-address decoding to complete the system. With the latches, the Module communicates with an IHU as 63 Digital Output signals (IHU to Module), 64 Digital Input signals (Module to IHU), five Analog Inputs (Module to IHU) and Module Power Control. **Figure 10** shows this configuration:

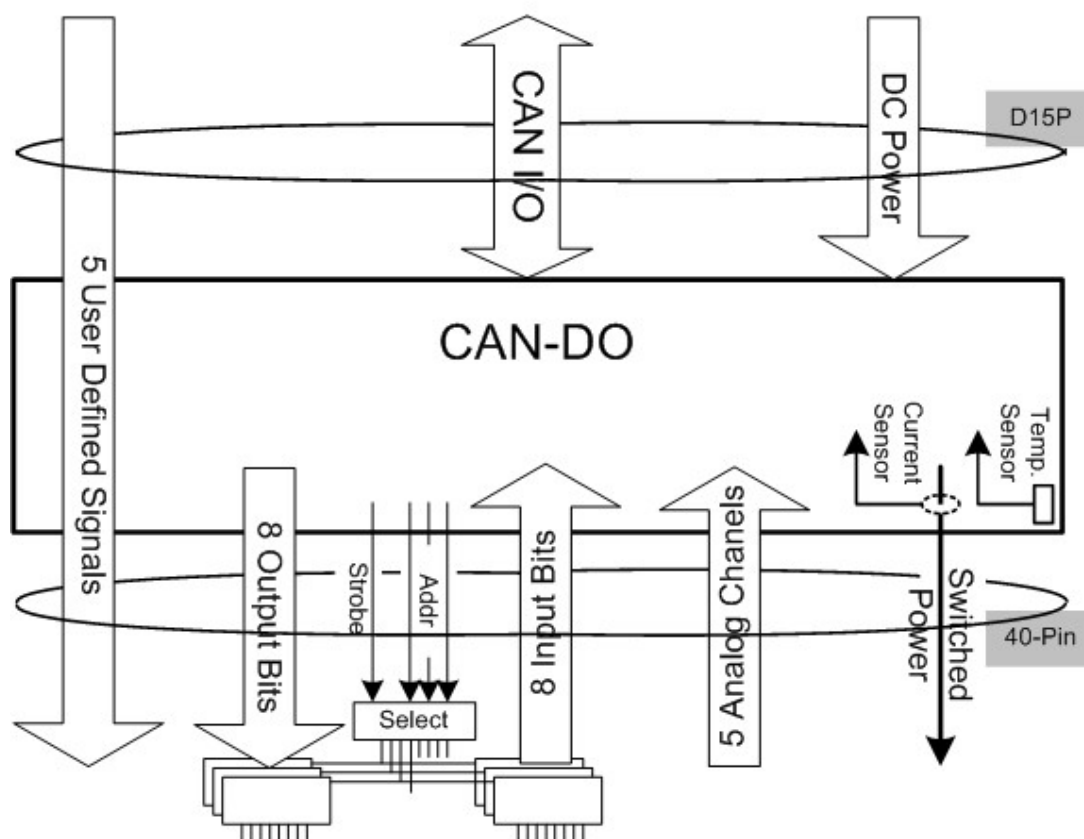


Figure 10 - Multiplex-mode Widget I/O Overview

The Widget provides non-switched 5V power at the 40-pin connector specifically to power the latches and latch-address decoding circuitry. This allows the 63 control lines to be placed in a desired state even though the Module is not powered on (or maintained in a given state while the module is power-cycled).

NOTE: This non-switched power for the external latch circuitry is not included in the current measurement of power going to the Module (current sensor) and should not be used for *any* other purpose in the Module.

In this Multiplex-mode configuration, an IHU can write all 63 output bits and power control, read the 64 input bits, the five Module-defined and 3 dedicated analog input channels (10-bit precision). It should be able to do this with all the Modules on the CAN bus every 20 milliseconds.

An IHU accomplishes this by placing the 63 output bit values and the power control value into a single 8-byte CAN packet, addressing it to the Module, marking it as a "configure" stream and sending the packet over the CAN bus.

Upon receipt of the "configure" packet the Widget adjusts Module power, if requested, then steps through the 8 latches, starting from zero, writing the output byte for this latch to the Module and reading the Input byte from the Module. (*NOTE: the read occurs after the output byte is written but while the Strobe is still asserted.*) After it has completed writing, then reading, each of the eight latches, it digitizes all eight analog channels (Temperature, Current, Current Bias, and 5 Module defined) and then (in a change from earlier versions of firmware) runs through the entire multiplex write/read sequence again. After this second multiplex effort, the Widget sends three 8-byte packets back to an IHU, which contain (1) 64 Digital inputs, (2) the first four Analog digitization results, and (3) the last four Analog results. *For precise packet descriptions, see: "Appendix C: CAN Message Specifications" later in this document.*

Rationale for two multiplex passes

The net effect of the two multiplex passes is to configure the Module on the first pass (ignoring the inputs). Allow the Module to react to the configuration during the ADC effort (6mSec) and then write the same control values to the Module (identical so not causing any change) then capture the post ADC inputs from the Module. It is the second set of inputs, which are sent to the IHU. Our early designers discovered that there is benefit to reducing the delay between "telemetry read" and "delivery to the IHU" to a minimum. By adding this second multiplex pass, we have reduced this delay.

The next section presents specific timing and pin out data for the Widget configured for Multiplex Mode.

Multiplex Pins and Timing

Measurements were taken during testing of a recent version of the widget firmware. **Table 2** shows timing of the behavior of the firmware with respect to the multiplexed I/O pins. The symbols presented in **Table 2** are shown in timing waveform diagrams **Figure 11 - Multiplex byte write (to Module from Widget)** and **Figure 12 - Multiplex byte read (from Module to Widget)**, which show the byte output, and byte input waveforms, respectively. The screen of the oscilloscope used to take the measurements is shown as **Figure 13 - Multiplex eight Latches being written/read** depicting all eight latches being accessed in sequence.

| <i>Symbol</i> | <i>Parameter</i> | <i>Limits</i> | | <i>Units</i> |
|--------------------|--|---------------|------------|--------------|
| | | Min | Max | |
| <i>Tw1</i> | Strobe Pulse Width | 2.235 | | uSec |
| <i>Tw2</i> | Strobe to Strobe Duration | 10.495 | | uSec |
| <i>Tsu1</i> | Latch Address Setup to Strobe assert | 2.275 | | uSec |
| <i>Tsu2</i> | Output Data set up to Strobe assert | 0.765 | | uSec |
| <i>Tsu3</i> | Strobe Assert set up to Input Data Valid | 0.500 | | uSec |
| <i>Th11</i> | Data valid hold to Strobe de-assert | 1.735 | | uSec |
| <i>(Not Shown)</i> | First Activity (Latch Address assert) to last Strobe de-assert | 78.0 | | uSec |

Table 2 - Measurements from released firmware v3.04

Table 2 NOTES:

1. Table is based on measurements of running released code Ver 3.04 dated Oct 2006.
2. CAN DO Widget Oscillator Frequency is 8MHz, T89C51CC01 is running in X2 Mode.
3. Above numbers are measured vs. calculated from the Osc. Freq.
(Need to be relative to ideal oscillator then can provide min/max based on Osc. tolerance.)
4. These lines are controlled by the T89C51CC01 micro-processor with a single instruction time of 750 nSec. (Given Note2), hence, the narrowest pulse width we can generate is 750 nSec. Our strobe pulse is wider than this since we are reading and storing the inputs during the pulse.

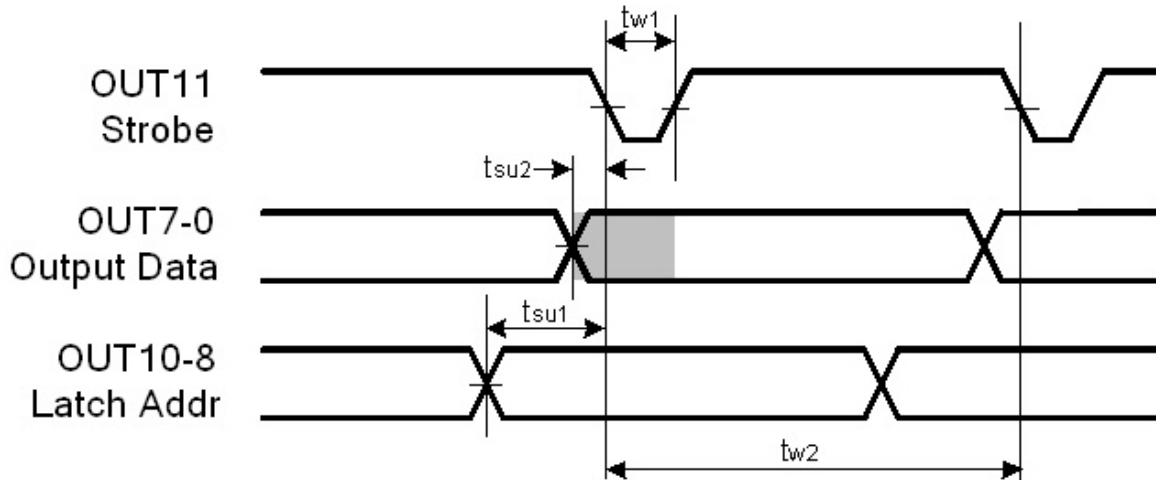


Figure 11 - Multiplex byte write (to Module from Widget)

In **Figure 11** we see that the widget first writes the latch address. It then writes the data byte for that specific latch. Then, finally, it asserts the Strobe. After a short interval, the widget then de-asserts the Strobe to complete the transaction. *NOTE: The strobe is defined as being active low due to the fact that our ATMEL T89C51CC01 comes out of reset with all the programmed I/O pins in a high state.*

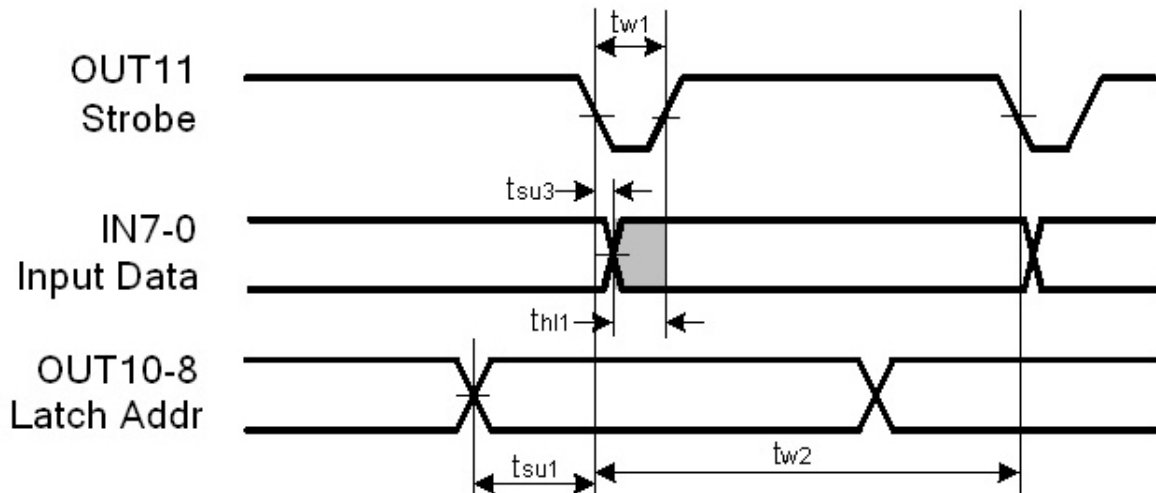


Figure 12 - Multiplex byte read (from Module to Widget)

In **Figure 12** we, again, see the Latch address being written first then the Strobe being asserted. A short time after the Strobe is asserted (~500 nSec.) the Data Input bus is read and stored in memory. The widget then de-asserts the Strobe to complete the transaction.

OPERATING MODES - MULTIPLEX

While we show **Figure 11 - Multiplex byte write (to Module from Widget)** and **Figure 12 - Multiplex byte read (from Module to Widget)** separately for clarity, they actually occur simultaneously.

This, then, is the full latch-control sequence (remember that this happens twice):

For each of the latch addresses 0-7, in sequence...
Write the latch address
Write the output data byte
Assert the strobe
Read the input data byte
Store the new input value
De-assert the Strobe
End for

The following picture shows an actual oscilloscope capture of this 8-latch multiplex sequence. (NOTE: the input data bus is not shown in the picture.)

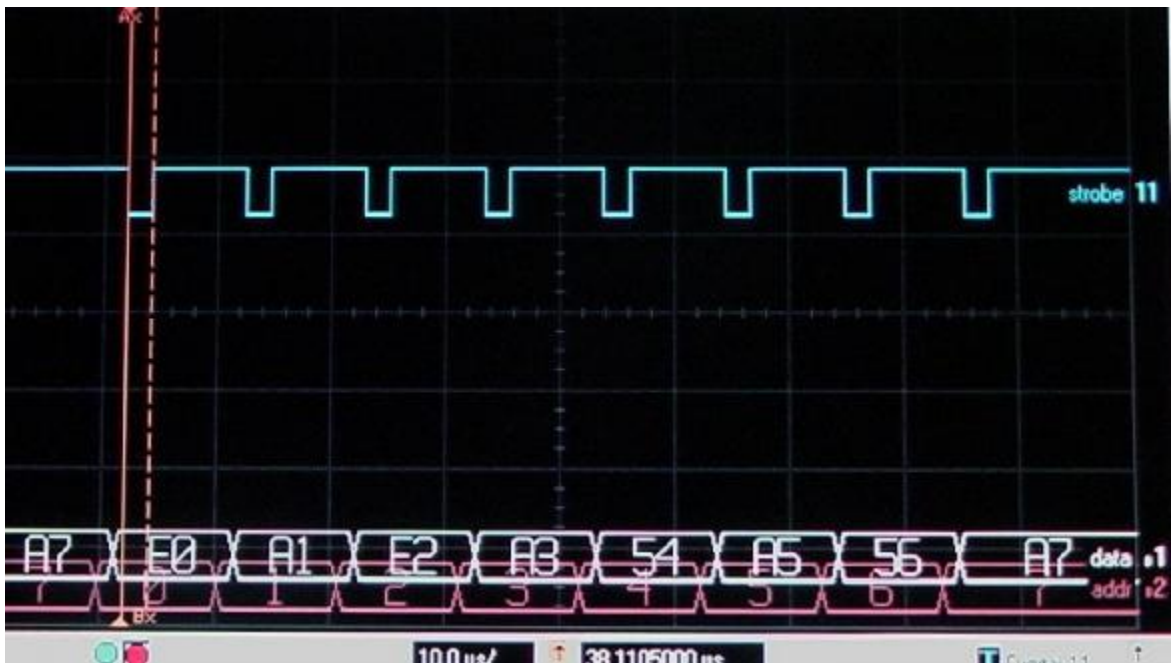


Figure 13 - Multiplex eight Latches being written/read

(Latch address in pink, data-out in white, and strobe in blue.)

Note: Agilent 54832D Oscilloscope used courtesy of [Agilent Technologies, Inc.](http://www.agilent.com) It is a 1GHz Mixed-signal Infiniium Oscilloscope having 4 analog channels and 16 Digital channels (which you see displayed in **Figure 13 - Multiplex eight Latches being written/read.**)

Sample interface Schematic

The schematic shown in **Figure 14** shows the multiplex address decoding, latching of one of the eight input channels, latching of one of the eight possible output channels and handling of the strobe.

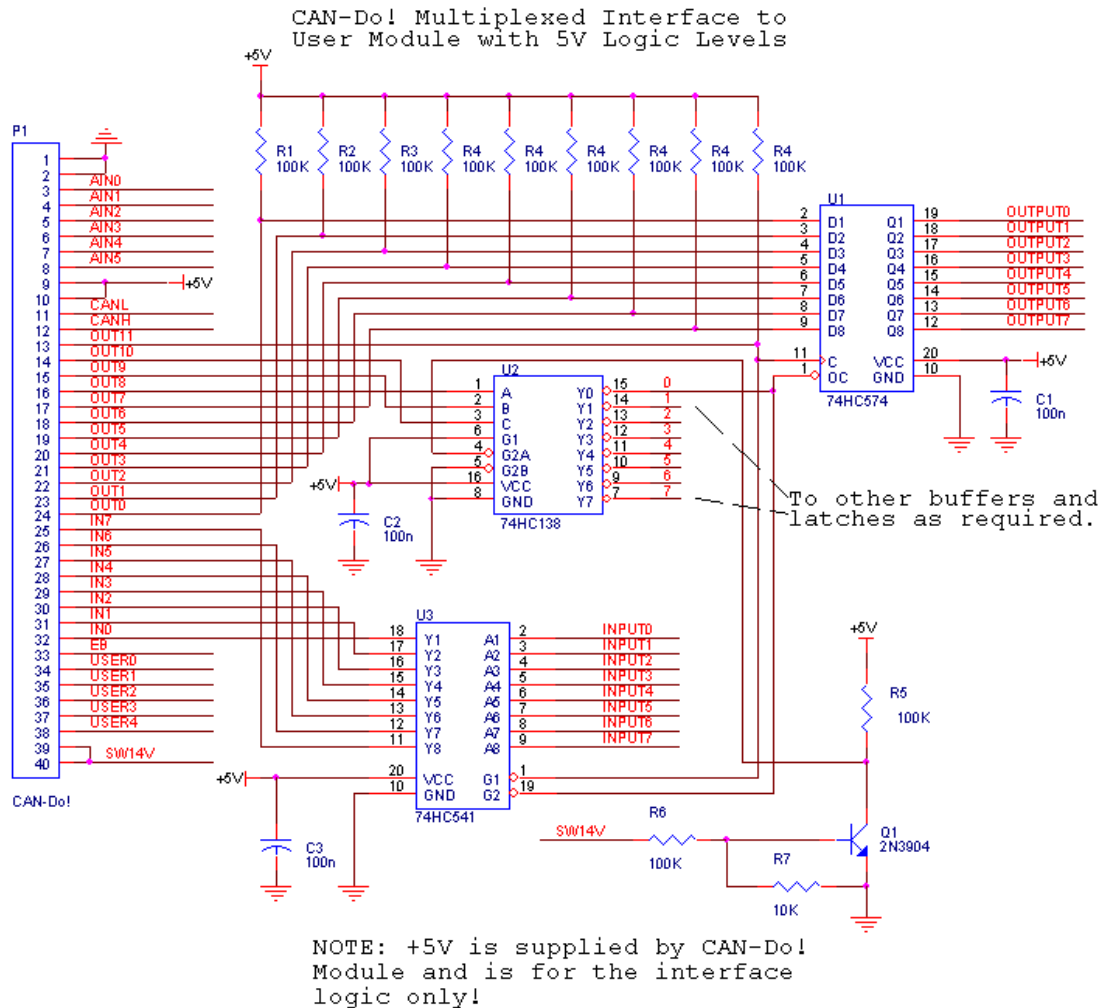


Figure 14 - Example multiplex decoding with 1 of 8 latch-pairs shown

Note that this is not a circuit recommendation; it is untested. It is only intended to provide clarity for the requirements of this interface:

- a) The user module must not load the CAN-Do! widget if the module is powered off.
- b) The user module must provide pull-ups to +5V on all CAN-Do! outputs that are used by the user module.
- c) The CAN-Do! widget 5V power is only for driving a few milliamperes at most.
- d) If the user module is powered off, all drive signals must be either tri-stated or driven LOW.

7: Byte-pipe Mode

Behavioral description and pin-outs

The CAN Widget, when configured for Byte-pipe mode has less general I/O than other modes, but provides a byte-wide input port and a byte-wide output port. These two ports operate independently of each other and are nearly independent of the normal "configure" communications with an IHU (See "Byte-pipe versus Module Power", below). **Figure 15** shows the byte-pipe configuration:

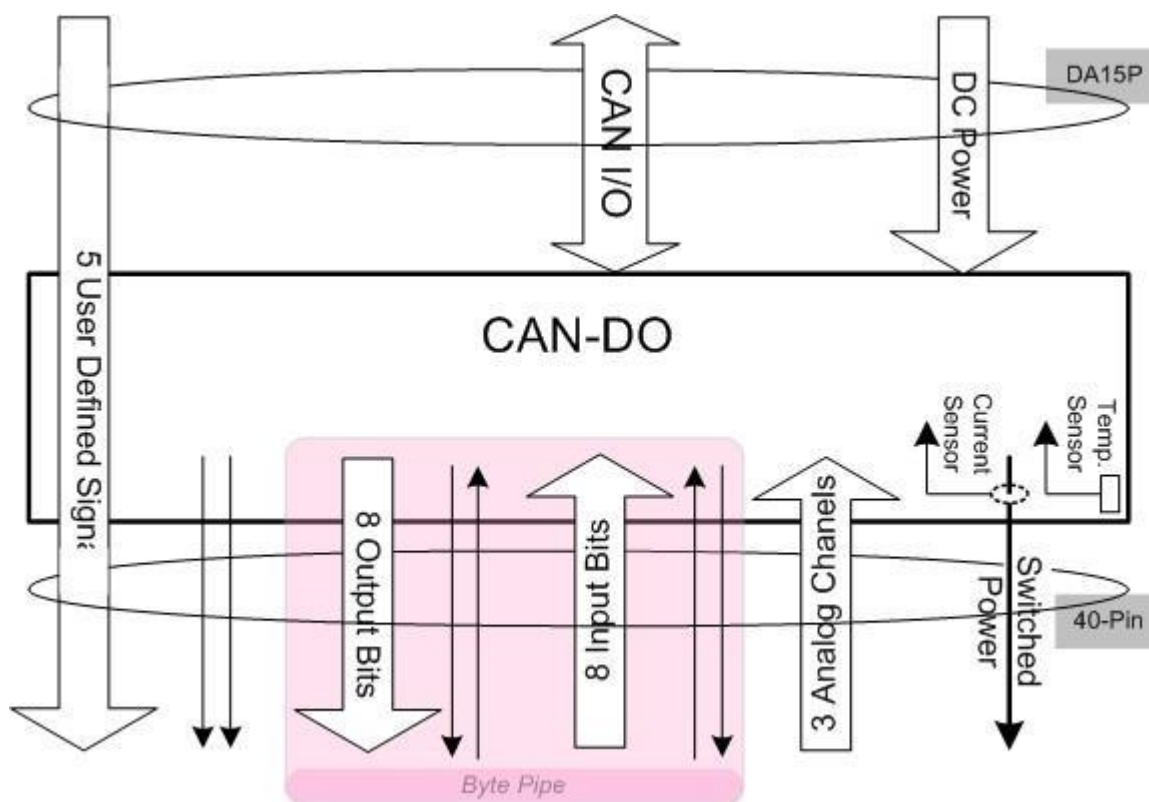


Figure 15 - Byte-pipe-mode Widget I/O Overview

Resources not consumed by the byte-pipe are two Digital Output signals (IHU to Module), three Analog Inputs (Module to IHU) plus Module Power Control which are read/written via "configure" communications with an IHU. Two types of communication take place with a Widget running in Byte-pipe mode: (1) "configure" traffic with an IHU, and (2) lower-priority data traffic with an IHU. All configure traffic on the CAN bus takes priority. The data traffic can only consume the remaining CAN bus bandwidth.

"Configure" Traffic

In Byte-pipe Mode, an IHU can write the 2 output bits and the power control and read the 3 Module defined and 3 dedicated analog input channels (10-bit precision). It should be able to do this with all the Modules on the CAN bus, irrespective of configured mode, every 20 milliseconds.

An IHU accomplishes this by placing the 2 output bit values and the power control value into a single 2-byte CAN packet, addressing it to the Module, marking it as a "configure" stream and sending the packet over the CAN bus.

Upon receipt of this "configure" packet the Widget adjusts Module power, if requested, sets the output bits appropriately, digitizes the 6 analog channels (Temperature, Current, Current Bias, and the 3 Module defined channels) and, finally, sends two 8-byte packets back to an IHU which contain (1) the first three Analog digitization results, and (2) the last three Analog results. *For precise packet descriptions, see: "Appendix C: CAN Message Specifications" later in this document.*

Byte-pipe Read/Write Traffic

The Byte-pipe functionality is separate from the "configure" functionality and consists of two independent halves: Input (from Module to IHU) and Output (from IHU to Module). These halves are conditioned by Module power. If power is applied, the Byte-pipes operate. If there is no power, they are idle.

The Byte-pipe Input channel waits for bytes to be written to the Widget by the Module. As bytes arrive, they are placed into CAN packets and sent to an awaiting IHU. The Widget, under normal conditions, constructs full 8-byte packets, so upon receipt of an eighth byte a packet is generated and sent. We do not want the Module electronics to be able to stop a Widget from functioning. In order to prevent this we have included special mechanisms to detect and recover from modules failing to work correctly. Under some circumstances, the Widget may send short packets (less than 8-bytes). *See the section below entitled "Transfer Stall Recovery" for a discussion of these special mechanisms and circumstances.*

The Output channel is idle until a Byte-pipe write CAN packet is received by the Widget. After it completely receives the packet, the eight bytes are written to the output channel one byte at a time, as fast as the Module acknowledges each until all eight bytes are transferred. *This output side is also protected and is subject to the mechanisms described in the section below entitled "Transfer Stall Recovery".*

Byte-pipe versus Module Power

While the Module Power is turned off there is effectively no device at the other end of the byte-pipe. Therefore, the Module-to-Widget side is ignored completely and the Widget to Module side is not driven. Should an IHU send byte-pipe write packets they will simply be ignored (the CAN packet receive is completed, but the data is ignored).

If the Module power is dropped while an input or output transfer is in progress the transfers are immediately halted in their current state. No further handshakes occur and all knowledge of pending transfers is lost. As the power is restored, the two halves are reset and are ready to begin transfers.

NOTE: The IHU firmware developer should take care to implement the upper level protocol with a byte-pipe Module such that the Module power is not manipulated unless there are no transfers in progress.

Byte-pipe power-on stabilization interval

For a short period after power is first applied by the Widget to the Module electronics the pipe-input handshake lines are ignored. This interval provides the Module with enough time to begin overt control of the pipe-handshake lines immediately after power-up and prevents the Widget from falsely detecting a pipe-input request during voltage changes on the input lines during the power-up of the Module electronics.

A precise indication of this delay is provided to the Module electronics by asserting the input ACK line immediately prior to powering on the Module. After a short delay (~ 4mSec, two input-timer expirations), the input ACK is de-asserted indicating that the Widget is now ready to accept input traffic. This delay indication, informs the Module electronics of the period during which the state of the input lines are completely ignored. The input ACK line was chosen as its natural purpose in the handshake protocol was already to be a holdoff to subsequent input REQuests. In this delay-indication use it serves the same function: the Module is to hold-off REQuesting any input transfers until the delay indication is removed (ACK is de-asserted).

Transfer Stall Recovery

Two mechanisms are provided to keep the Widget in sync with the Module: (1) the Byte-pipe halves can be reset, and (2) a transfer that takes too long will be truncated.

The Byte-pipe is stateful. Should a Module get out of sync with the Widget Byte-pipe, an IHU can command the Widget to reset the Byte-pipe (the Input and Output halves are reset at the same time.) See: "**Appendix C: CAN Message Specifications**" configure command packet for details on this reset request.

The design of the Byte-pipe protocol allows for maximum range in transfer speeds and little overhead in firmware in the Widget. While there is a maximum rate at which the Widget can handle transfers, the only definition of minimum is the timer mechanism we provide which will abort any transfer exceeding the specified (generous) amount of time for an 8-byte transfer.

On the input side of things, timer expiration means that an Input byte request did not happen in the desired amount of time. **The bytes received, so far, are placed in an outgoing short packet and the packet sent.** This then clears the input side so that the building of a new 8-byte packet is started with the next byte received. If the packet received by an IHU is shorter than 8-bytes, it can know that the packet was terminated by a transfer timeout. Additionally, a count of these input-timer expiration events is maintained and is shipped in the AN03 Module Status packet. See: the "**Appendix C: CAN Message Specifications**" for details of this packet.

On the Output side, timer expiration means that an Output request was not acknowledged in time. This means the Module is no longer paying attention to the interface. **In this situation, further writes to the Module by the Widget are ceased and the rest of the bytes (those not yet transferred) are dropped.** This clears the Output side so that the next packet received from an IHU starts the first of the next eight transfers. As on the input side, a count of these output-timer expiration events is maintained and is shipped in the AN03 Module Status packet. See: the "**Appendix C: CAN Message Specifications**" for details of this packet.

NOTE1: The Module designer should take care to implement the upper level protocol using 8-byte transfers exclusively.

NOTE2: The Module designer may wish to use one of the Digital outputs of the "Configure" request to implement a reset line to the Module. If this is used, the IHU logic can reset the Module and the Byte-pipe logic in the same "Configure" packet.

The next section presents specific timing and pin out data for a Widget configured for Byte-pipe Mode.

Byte-pipe Pins and Timing

Latest versions of the widget firmware support Byte-pipe mode. An upcoming revision of this document will contain timing specifications and oscilloscope pictures. Meanwhile, pin assignments have been made and are presented in this section. Likewise, the handshake protocol has been determined and is described herein.

The following diagrams and tables refer to sender and receiver. For byte-pipe output (IHU to Module) the sender is the widget and the receiver is the module. For byte-pipe input (Module to IHU), this is reversed, the sender is the module, while the receiver is the widget.

The I/O lines have been assigned purposes. **Table 3** shows this assignment:

| <i>Purpose</i> | <i>Description</i> | <i>40-Pin Connector</i> |
|--------------------|---|-------------------------|
| <i>Out[0-7]</i> | Data byte write to Module (from Widget) | OUT0-7 |
| <i>OUT Request</i> | Byte is ready to be read by Module | OUT10 |
| <i>OUT Ack</i> | Module has read byte | AIN0 |
| <i>In[0-7]</i> | Data byte read from Module (to Widget) | IN0-7 |
| <i>IN Request</i> | Byte is ready to be read by Widget | AIN4 |
| <i>IN Ack</i> | Widget has read byte | OUT11 |
| <i>(open)</i> | <i>Available for use by Module Designer</i> | OUT8 |
| <i>(open)</i> | <i>Available for use by Module Designer</i> | OUT9 |
| <i>(open)</i> | <i>Available for use by Module Designer</i> | AIN1 |
| <i>(open)</i> | <i>Available for use by Module Designer</i> | AIN2 |
| <i>(open)</i> | <i>Available for use by Module Designer</i> | AIN3 |

Table 3 - Byte-pipe Pin Assignments

NOTE for **Table 3**: The AIN0 and AIN4 lines are digital lines as are OUT10 and OUT11. However, the AINx lines have additional circuitry conditioning them for use as analog inputs. While used as byte-pipe handshake, the behavior of the capacitors on these lines dominate by slowing the handshake signal's rise and fall times. With the 0.1 uF CP2 in place, the effective transfer rate is about 7K Bytes per second. With CP2 removed, the speed climbs to 90+K Bytes per second. Since the module temperature sensor AIN6 is conditioned by CP2, the developer should not simply remove CP2. Instead, CP2 should be replaced by two 0.1 uF 0603 capacitors placed across pins 1 & 8 and pins 3 & 6. In this way AIN2 and AIN6 have the proper capacitance while our analog inputs -turned digital- can now be run at full speed.

OPERATING MODES - BYTE-PIPE

The byte transfer handshake is simple. **Figure 16** shows the coordination between the Sending and Receiving activities:

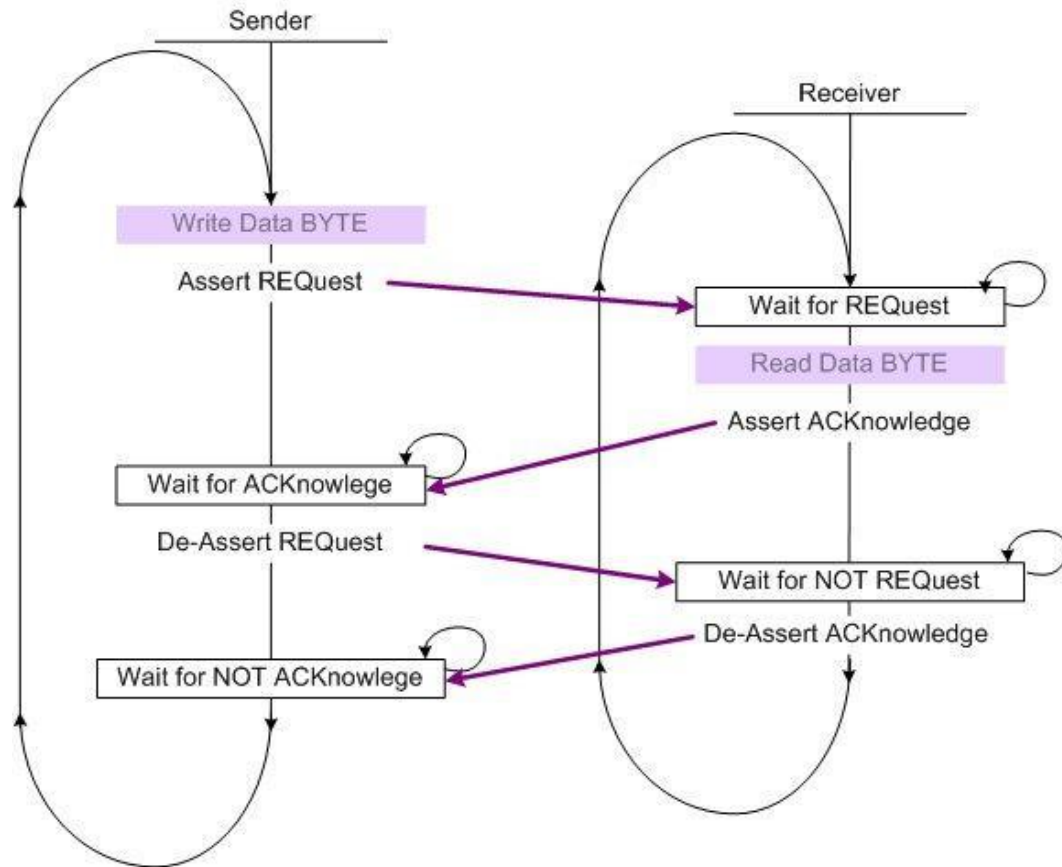


Figure 16 - Coordination between sender and receiver

Our output pins are high coming out of reset for this part. To reduce glitches or inadvertent signaling, our signals are defined as de-asserted when high (e.g., when coming out of reset). Given this, an example waveform for the protocol is shown in **Figure 17**.

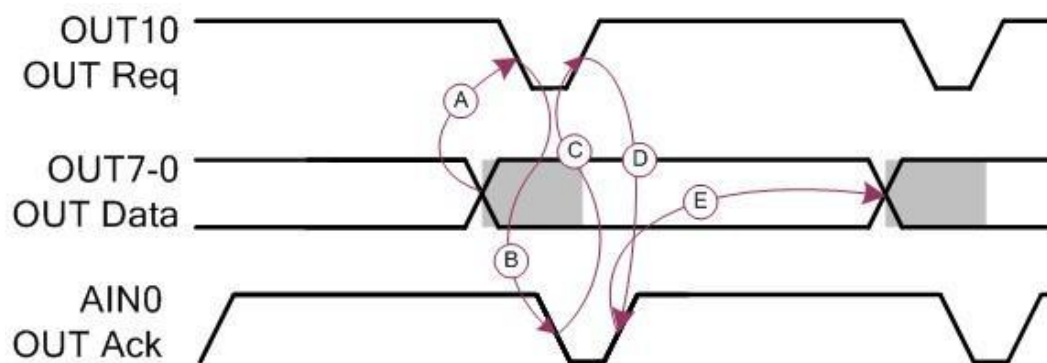


Figure 17 - Example Byte-pipe Handshake

In the waveform shown in **Figure 17** we see the following events:







- a) **The sender writes DATA then asserts REQ.**
- b) **The receiver sees the REQ, reads the DATA and asserts ACK.**
- c) **The sender sees the ACK, and then de-asserts REQ.**
- d) **The receiver sees REQ go away, and then de-asserts ACK.**
- e) **The sender waits for ACK to go away before starting with the next byte.**

A: CAN-DO! Schematics





The schematics occupy the next two pages.

You may also refer to the schematics posted at our web site:
(<http://can-do.moraco.info/doc/canV16.pdf>)

Page 1 contains:





-  The ATMEL controller
-  CAN bus interface circuitry
-  Digital input signal conditioning
-  Digital output signal conditioning
-  Analog voltage reference
-  Mode and address decoding logic

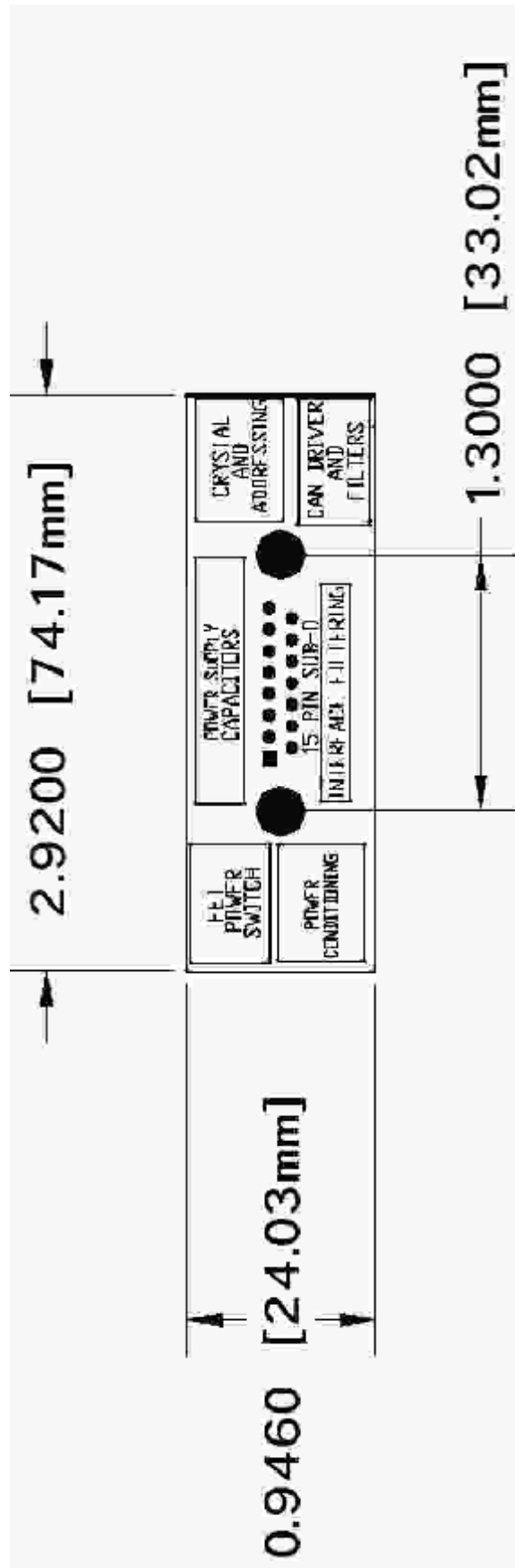
Page 2 contains:

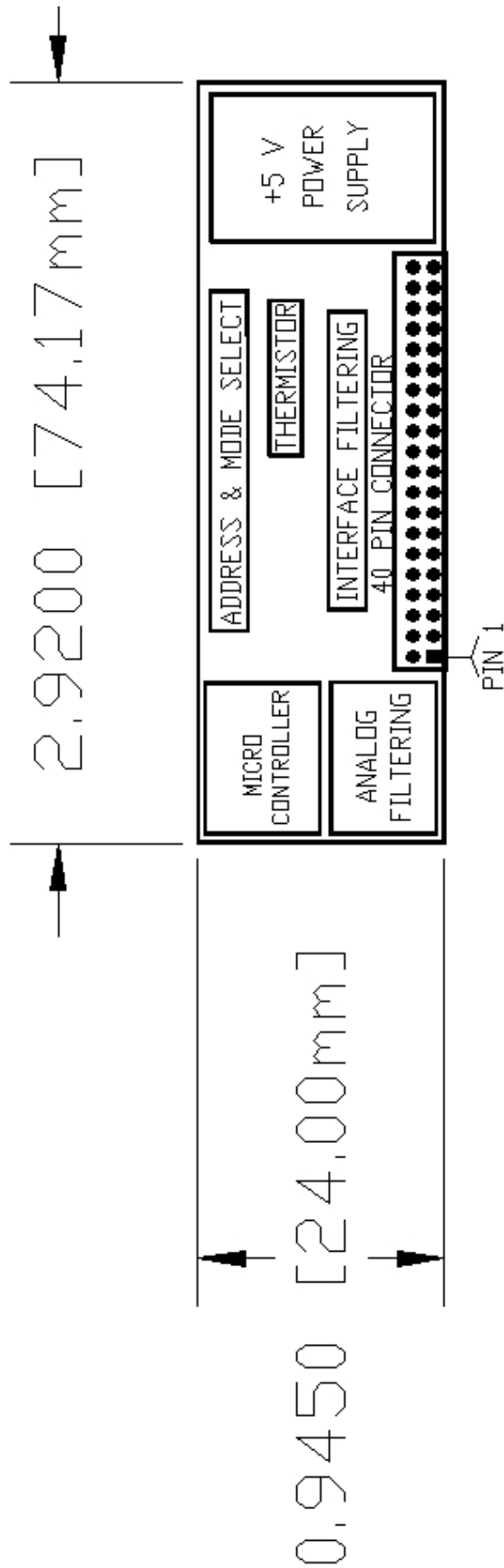
-  Analog input signal conditioning
-  Power conditioning
-  The D15P and 40-pin connectors
-  Three widget-dedicated analog sensors: temperature, current and current bias

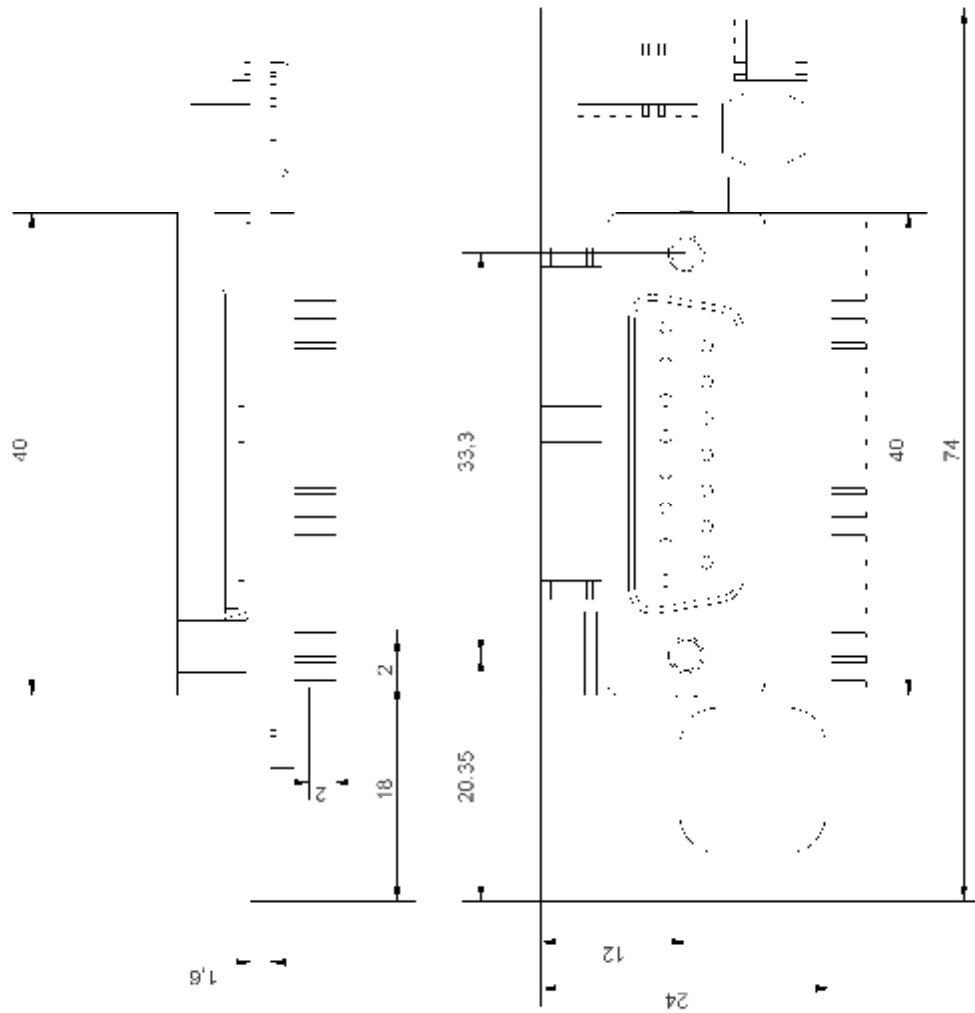
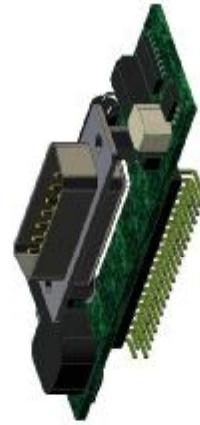
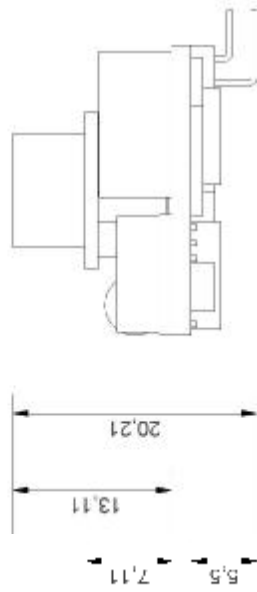
B: Mechanical Drawings

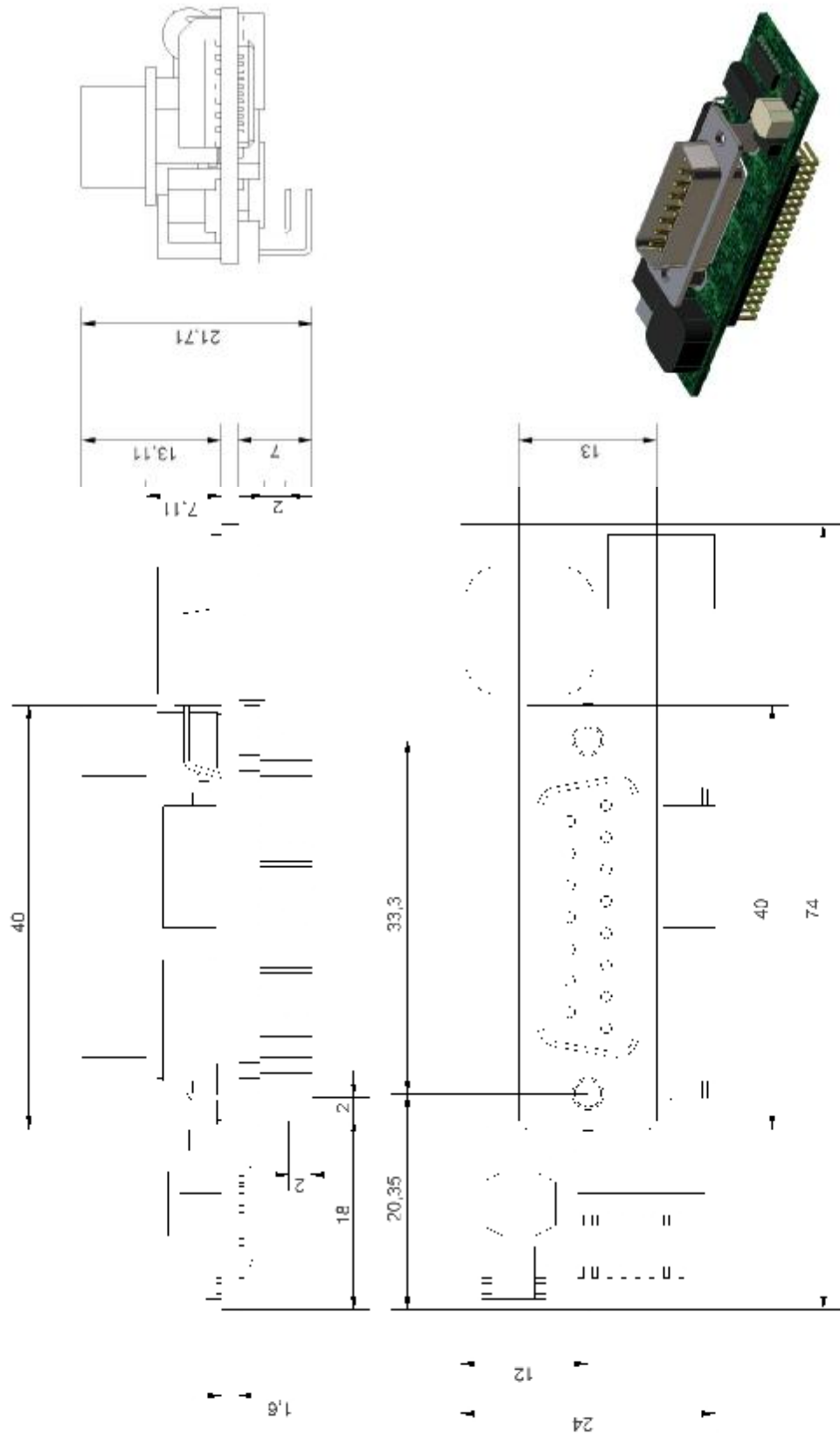
Four mechanical drawings are presented in this section, a page each:

-  Front side layout and overall dimensions
-  Back-side layout and overall dimensions
-  Detailed dimensions with 40-pin connector extending down from bottom of board
-  Detailed dimensions with 40-pin connector extending up











C: CAN Message Specifications

This message definition implements a command/status bus used to communicate between an IHU and per-Module CAN Widgets on the spacecraft. The intent is that the IHU is communicating with the Module attached to the Widget, through the Widget, so the narratives below here speak mostly as if the Widget is transparent in this communication.

CAN Module Addressing

For the AMSAT CAN Widgets the 11 bit CAN Bus Message ID space is structured as follows:

| | |
|----------------------------------|--|
| 10 9 8 7 6 5 4 3 2 1 0 | |
| + | These bits represent the address of |
| +--- | the Widget. They are set by shorting |
| +----- | solder-pads on the Widget. |
| +----- | IHU(s) will have the lowest possible |
| +----- | address(es) - 0x01. The lowest |
| +----- | valid CAN message ID. Also the |
| | highest priority address on CAN BUS. |
| v v v v | |
| | *** IHU to Module *** |
| 0 0 0 0 0 ----- | - 0x00MM Module Configure Command |
| 0 0 0 0 1 ----- | - 0x04MM reserved |
| 1 0 0 0 1 ----- | - 0x44MM Byte-pipe data to write to Module |
| 0 0 0 1 0 ----- | - 0x08MM reserved |
| 0 0 0 1 1 ----- | - 0x0CMM reserved |
| 0 0 1 0 0 ----- | - 0x10MM reserved |
| 0 0 1 0 1 ----- | - 0x14MM Module Census Query |
| 0 0 1 1 0 ----- | - 0x18MM Module Read Counters |
| 0 0 1 1 1 ----- | - 0x1CMM (deprecated) |
| | |
| | *** Module to IHU *** |
| 0 1 0 0 0 ----- | - 0x20MM Module Status IN08 (Mux. Mode only) |
| 0 1 0 0 1 ----- | - 0x24MM Module Status AN03 |
| 0 1 0 1 0 ----- | - 0x28MM Module Status AN47 |
| 0 1 0 1 1 ----- | - 0x2CMM reserved |
| 1 1 0 1 1 ----- | - 0x6CMM Byte-pipe data read from Module |
| 0 1 1 0 0 ----- | - 0x30MM Device Info Page Response |
| 1 1 1 0 0 ----- | - 0x70MM ERAM-Dump Response |
| 0 1 1 0 1 ----- | - 0x34MM Module Census Response |
| 0 1 1 1 0 ----- | - 0x38MM Module Read Counters Response |
| 0 1 1 1 1 ----- | - 0x3cMM (deprecated) |
| | |
| | |
| + ----- | This bit identifies messages |
| | that adhere to this protocol. |
| | A zero represents a hi-priority |
| | message, which implements this |
| | protocol. A one represents |
| | a low priority bulk data message . |

This can be thought of as **PSSSSAAAAAA** with the following legend:

- P - Protocol (0=this, 1=low priority bulk data) [mask 0x400]
- SSSS - Stream ID (0-15 shifted left 6 bits) [mask 0x3C0]
- AAAAAA - Widget Address (1-63, 0=illegal) [mask 0x03F]

NOTE: by convention our test software (UHU, CDNC) both leave 1,2 for IHUs and 3-9 as other controller addresses. This allows for the test software and the IHUs to all be on the CAN bus at the same time (in the 1-9 range). In this form the test software can be logging/controlling traffic and we can walk up to a test CAN bus and plug in an IHU without re-addressing the Widgets on the bus or moving the test controller address.

Standard Mode CAN Messages

The Standard mode gets its name from the fact that this is expected to be the most prevalent mode on the Spacecraft.

The messages in this section represent two types of transactions on the CAN bus:

1. Configure the Module, which then replies with all of its sensor values (digital and analog).
2. Query the CAN bus to see which Modules are present. *This is likely used only during Module development, and ground testing.*

The 0x00MM Module Configure Command and the 0x24MM and 0x28MM response packets implement the Standard mode configure transaction.

0x00MM Module Configure Command

[2-BYTE PAYLOAD]

This command packet is used to send digital output values to the Module from an IHU. It is also used to turn the Module on and off.

The Standard mode Module Configure Command packet consists of 2 bytes. Unused bits are ignored (their value does not matter, zero's are shown below.) The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | Out7..Out0 |
| 1.7 | PwrCtl: 1=Module power ON, 0=Module power OFF |
| 1.6-4 | reserved (0's) |
| 1.3-0 | Out11..Out8 |

NOTE: This shorter configure packet is an intentional design decision. By this packet being the most prevalent configure packet on the CAN bus and by its being shorter, we free up a significant amount of bus bandwidth for other uses.

A Standard mode Widget must send two packets in response to receiving this packet. They are 0x24MM and 0x28MM.

0x24MM Module Status AN03

[8-BYTE PAYLOAD]

This is the first of two packets sent in response to a Module Configure Command to the IHU by a Standard-mode Widget.

This packet contains the 10-bit values for Analog Channels 0-3. It also carries four of the eight digital inputs (IN0-IN3) and additional feedback* values.

[See 0x28MM - data for Analog/Digital Input Channels 4-7]

The 0x24MM Module Status AN03 response packet consists of 8 bytes. Unused bits will be zero and should be ignored. The packet content is defined as follows:

APPENDIX C: CAN MESSAGE SPECS

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | Analog Channel 0 bits 7..0 |
| 1.7 | Digital IN0 |
| 1.6-2 | reserved (0's) |
| 1.1-0 | Analog Channel 0 bits 9..8 |
| 2.7-0 | Analog Channel 1 bits 7..0 |
| 3.7 | Digital IN1 |
| 3.6-2 | reserved (0's) |
| 3.1-0 | Analog Channel 1 bits 9..8 |
| 4.7-0 | Analog Channel 2 bits 7..0 |
| 5.7 | Digital IN2 |
| 5.6-3 | reserved (0's) |
| 5.2 | * Module Switched Power State (1=on, 0=off) |
| 5.1-0 | Analog Channel 2 bits 9..8 |
| 6.7-0 | Analog Channel 3 bits 7..0 |
| 7.7 | Digital IN3 |
| 7.6 | * CHECKSUM FLASH: 1=valid, 0=invalid |
| 7.5 | * CHECKSUM ERAM: 1=valid, 0=invalid |
| 7.4 | * CHECKSUM RAM: 1=valid, 0=invalid |
| 7.3-2 | * Mode bits (M1,M0) showing jumpered mode |
| 7.1-0 | Analog Channel 3 bits 9..8 |

*NOTE: * denotes Widget state feedback to control software. (This status is available for use by control software for diagnostic reasons, positive control feedback, etc.)*

0x28MM Module Status AN47

[8-BYTE PAYLOAD]

This is the second of two packets sent in response to a Module Configure Command to an IHU by a Standard-mode Widget.

This packet contains the 10-bit values for Analog Channels 4-7. It also carries the second four of the eight digital inputs (IN4-IN7) [See 0x24MM - data for Analog/Digital Input Channels 0-3]

The 0x28MM Module Status AN47 response packet consists of 8 bytes. Unused bits will be 0 and should be ignored. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|----------------------------|
| 0.7-0 | Analog Channel 4 bits 7..0 |
| 1.7 | Digital IN4 |
| 1.6-2 | reserved (0's) |
| 1.1-0 | Analog Channel 4 bits 9..8 |
| 2.7-0 | Analog Channel 5 bits 7..0 |
| 3.7 | Digital IN5 |
| 3.6-2 | reserved (0's) |
| 3.1-0 | Analog Channel 5 bits 9..8 |
| 4.7-0 | Analog Channel 6 bits 7..0 |
| 5.7 | Digital IN6 |
| 5.6-2 | reserved (0's) |
| 5.1-0 | Analog Channel 6 bits 9..8 |
| 6.7-0 | Analog Channel 7 bits 7..0 |
| 7.7 | Digital IN7 |
| 7.6-2 | reserved (0's) |

APPENDIX C: CAN MESSAGE SPECS

7.1-0 Analog Channel 7 bits 9..8

0x14MM Module Census Query

[BROADCAST, 0-BYTE PAYLOAD]

NOTE: This message was intended to be used during Module development, and ground testing; however, there is no reason why the system developer could not use this in flight if deemed useful.

This message is used to poll a bus to see which Widgets are powered up/present. Since this is a broadcast, the address part of the 11-bit ID contains the sender's address, not the usual address of the intended receiver.

Widgets receiving this message should reply using their own address (in the address part of the message ID) with a 0x34MM response packet properly filled in. At first, this seems weird, but it prevents collisions on the CAN bus (all replying to same address) and provides for responses arriving to the requesting device in natural widget-priority order. The querying device simply listens for all 0x34MM packets irrespective of the "sending" address.

A device will send this packet then receive responses for a fixed amount of time thereafter (15 mSec should be sufficient for 63 devices on a bus.) After this time expires, it stops listening and assumes all powered-up devices have responded.

| Message Bits | Purpose |
|---|---------|
| ----- | |
| *** There is NO message content (payload) for this packet *** | |
| ----- | |

0x34MM Module Census Response

[8-BYTE PAYLOAD]

A Widget constructs and sends one of these packets whenever it receives a 0x14MM Module Census Query packet. The Widget must address this packet to its own address. The Querying device is listening for all 0x34MM packets ignoring the address field (the MM part).

The 0x34MM Module Census response packet consists of 8 bytes. Unused bits will be zero. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|--|
| ----- | |
| 0.7-6 | Module Mode |
| 0.5-0 | Module Address |
| 1.7-0 | reserved (DEBUG RAM Checksum MsByte DEBUG) |
| 2.7-0 | reserved (DEBUG RAM Checksum LsByte DEBUG) |
| 3.7-0 | reserved (DEBUG ERAM Checksum MsByte DEBUG) |
| 4.7-0 | reserved (DEBUG ERAM Checksum LsByte DEBUG) |
| 5.7-0 | reserved (DEBUG EEPROM Checksum MsByte DEBUG) |
| 6.7-0 | reserved (DEBUG EEPROM Checksum LsByte DEBUG) |
| 7.7-4 | Widget Firmware Version (Major revision nbr) |
| 7.3-0 | Widget Firmware Version (Minor revision nbr) |
| ----- | |

Multiplex Mode CAN Messages

The Multiplexed mode gets its name from the fact that the 63 outputs and the 64 inputs are multiplexed over simple 8-bit ports and must be latched by the Module. The input and output ports share 3 latch address lines and a single strobe line.

The messages in this section represent two types of transactions on the CAN bus:

1. Configure the Module, which then replies with all of its sensor values (digital and analog).
2. Query the CAN bus to see which Modules are present. *This is used during Module development, and ground testing, only.*

The **0x00MM Module Configure Command** and the **0x20MM**, **0x24MM** and **0x28MM response packets** implement the Multiplexed mode configure transaction.

0x00MM Module Configure Command

[8-BYTE PAYLOAD]

This command packet is used to send digital output values to the Module from an IHU. It is also used to turn the Module on and off.

The Multiplexed mode Module Configure Command packet consists of 8 bytes. There are no unused bits. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | Output Latch 0 bits 7..0 |
| 1.7-0 | Output Latch 1 bits 7..0 |
| 2.7-0 | Output Latch 2 bits 7..0 |
| 3.7-0 | Output Latch 3 bits 7..0 |
| 4.7-0 | Output Latch 4 bits 7..0 |
| 5.7-0 | Output Latch 5 bits 7..0 |
| 6.7-0 | Output Latch 6 bits 7..0 |
| 7.7 | PwrCtl: 1=Module power ON, 0=Module power OFF |
| 7.6-0 | Output Latch 7 bits 6..0 (There is NO bit 7 of latch 7) |

NOTE: this is setup specifically so that the byte offset into the message corresponds to the latch address.

NOTE2: Bit 7 of Latch 7 will always be written as a one to the Module.

A Multiplexed mode Widget must send three packets in response to receiving this packet. They are 0x20MM, 0x24MM and 0x28MM.

0x20MM Module Status IN08

[8-BYTE PAYLOAD]

This is the first of three packets sent in response to a Module Configure Command to an IHU by a Multiplex-mode Widget.

This packet carries the 64 digital input values from the Module to the IHU.

[See 0x24MM Module Status AN03 and 0x28MM Module Status AN47 for the analog channels.]

APPENDIX C: CAN MESSAGE SPECS

The Module Status IN08 response packet consists of 8 bytes. There are no unused bits. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|-------------------------|
| 0.7-0 | Input Latch 0 bits 7..0 |
| 1.7-0 | Input Latch 1 bits 7..0 |
| 2.7-0 | Input Latch 2 bits 7..0 |
| 3.7-0 | Input Latch 3 bits 7..0 |
| 4.7-0 | Input Latch 4 bits 7..0 |
| 5.7-0 | Input Latch 5 bits 7..0 |
| 6.7-0 | Input Latch 6 bits 7..0 |
| 7.7-0 | Input Latch 7 bits 7..0 |

NOTE: this is setup specifically so that the byte offset into the message corresponds to the latch address.

0x24MM Module Status AN03

[8-BYTE PAYLOAD]

This is the second of three packets sent in response to a Module Configure Command to an IHU by a Multiplex-mode Widget.

This packet contains the 10-bit values for Analog Channels 0-3. with additional feedback* values [See 0x28MM for Analog Channels 4-7 and 0x20MM for the Digital Inputs.]

The 0x24MM Module Status AN03 response packet consists of 8 bytes. Unused bits will be 0 and should be ignored. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | Analog Channel 0 bits 7..0 |
| 1.7-2 | reserved (0's) |
| 1.1-0 | Analog Channel 0 bits 9..8 |
| 2.7-0 | Analog Channel 1 bits 7..0 |
| 3.7-2 | reserved (0's) |
| 3.1-0 | Analog Channel 1 bits 9..8 |
| 4.7-0 | Analog Channel 2 bits 7..0 |
| 5.7-3 | reserved (0's) |
| 5.2 | * Module Switched Power State (1=on, 0=off) |
| 5.1-0 | Analog Channel 2 bits 9..8 |
| 6.7-0 | Analog Channel 3 bits 7..0 |
| 7.7 | reserved (0's) |
| 7.6 | * CHECKSUM FLASH: 1=valid, 0=invalid |
| 7.5 | * CHECKSUM ERAM: 1=valid, 0=invalid |
| 7.4 | * CHECKSUM RAM: 1=valid, 0=invalid |
| 7.3-2 | * Mode bits (M1,M0) showing jumpered mode |
| 7.1-0 | Analog Channel 3 bits 9..8 |

*NOTE: * denotes Widget state feedback to control software. (This status is available for use by control software for diagnostic reasons, positive control feedback, etc.)*

0x28MM Module Status AN47

[8-BYTE PAYLOAD]

This is the 3rd of three packets sent in response to a Module Configure Command to an IHU by a Multiplex-mode Widget.

This packet contains the 10-bit values for Analog channels 4-7.
 [See 0x24MM for Analog Channels 0-3 and 0x20MM for the Digital Inputs.]

The 0x28MM Module Status AN47 response packet consists of 8 bytes. Unused bits will be zero and should be ignored. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|----------------------------|
| 0.7-0 | Analog Channel 4 bits 7..0 |
| 1.7-2 | reserved (0's) |
| 1.1-0 | Analog Channel 4 bits 9..8 |
| 2.7-0 | Analog Channel 5 bits 7..0 |
| 3.7-2 | reserved (0's) |
| 3.1-0 | Analog Channel 5 bits 9..8 |
| 4.7-0 | Analog Channel 6 bits 7..0 |
| 5.7-2 | reserved (0's) |
| 5.1-0 | Analog Channel 6 bits 9..8 |
| 6.7-0 | Analog Channel 7 bits 7..0 |
| 7.7-2 | reserved (0's) |
| 7.1-0 | Analog Channel 7 bits 9..8 |

0x14MM Module Census Query

[BROADCAST, 0-BYTE PAYLOAD]

NOTE: This message was intended to be used during Module development, and ground testing; however, there is no reason why the system developer could not use this in flight if deemed useful.

This message is used to poll a bus to see which Widgets are powered up/present. Since this is a broadcast, the address part of the 11-bit ID contains the sender's address, not the usual address of the intended receiver.

Widgets receiving this message reply using their own address (in the address part of the message Id) with a 0x34MM response packet properly filled in. At first, this seems weird, but it prevents collisions on the CAN bus (all replying to same address) and provides for responses arriving to the requesting device in natural widget-priority order. The querying device simply listens for all 0x34MM packets irrespective of the "sending" address.

A device will send this packet then receive responses for a fixed amount of time thereafter (15 mSec should be sufficient for 63 devices on a bus.) After this time expires, it stops listening and assumes all powered-up devices have responded.

| Message Bits | Purpose |
|---|---------|
| *** There is NO message content (payload) for this packet *** | |

0x34MM Module Census Response

[8-BYTE PAYLOAD]

A Widget constructs and sends one of these packets whenever it receives a 0x14MM Module Census Query packet. The widget must address this packet to its own address. The Querying device is listening for all 0x34MM packets ignoring the address field (the MM part).

The 0x34MM Module Census response packet consists of 8 bytes. Unused bits will be zero. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-6 | Module Mode |
| 0.5-0 | Module Address |
| 1.7-0 | reserved (DEBUG RAM Checksum MsByte DEBUG) |
| 2.7-0 | reserved (DEBUG RAM Checksum LsByte DEBUG) |
| 3.7-0 | reserved (DEBUG ERAM Checksum MsByte DEBUG) |
| 4.7-0 | reserved (DEBUG ERAM Checksum LsByte DEBUG) |
| 5.7-0 | reserved (DEBUG EEPROM Checksum MsByte DEBUG) |
| 6.7-0 | reserved (DEBUG EEPROM Checksum LsByte DEBUG) |
| 7.7-4 | Widget Firmware Version (Major revision nbr) |
| 7.3-0 | Widget Firmware Version (Minor revision nbr) |

Byte-pipe Mode CAN Messages

The Byte-pipe mode gets its name from the fact that this mode implements a new byte-wide bi-directional transfer mechanism (pipe) in addition to the Module configure mechanism.

The messages in this section represent four types of transactions on the CAN bus:

1. Configure the Module, which then replies with all of its sensor values (digital and analog). *Optionally, the configure message can request a reset of the pipe-state machines and request clearing of the pipe event counters.*
2. Byte-pipe Write to Module.
3. Byte-pipe Read from Module.
4. Query the CAN bus to see which Modules are present. *This is used during Module development, and ground testing, only.*

The **0x00MM Module Configure Command**, the **0x24MM** and **0x28MM response packets** along with the **0x04MM Byte-pipe Data Write** and **0x2CMM Byte-pipe Data Read** packets implement the Byte-pipe mode.

0x00MM Module Configure Command

[2-BYTE PAYLOAD]

This command packet is used to send digital output values to the Module from an IHU. It is also used to turn the Module on and off.

The Byte-pipe mode Module Configure Command packet consists of 2 bytes. Unused bits are ignored (their value does not matter, zero's are shown below.) The packet content is defined as follows:

Message Bits Purpose

| Message Bits | Purpose |
|--------------|--|
| 0.7 | ResetCtl: 1-Reset Byte-pipe (IN & OUT), 0-Do not reset |
| 0.6 | ClrCounters: 1-Clear pipe-event counters, 0-Do not clear |
| 0.5-0 | reserved (0's) |
| 1.7 | PwrCtl: 1-Module power ON, 0-Module power OFF |
| 1.6-2 | reserved (0's) |
| 1.1-0 | Out9..Out8 |

NOTE this packet contains two diagnostic controls unique to byte-pipe mode. These reset both of the input and output byte-pipe halves without cycling power at the module and clear the pipe-event counters.

A Byte-pipe mode Widget must send two packets in response to receiving this packet. They are 0x24MM and 0x28MM.

0x24MM Module Status AN03

[8-BYTE PAYLOAD]

This is the first of two packets sent in response to a Module Configure Command to the IHU by a Byte-pipe-mode Widget.

APPENDIX C: CAN MESSAGE SPECS

This packet contains the 10-bit values for Analog Channels 1-3 with additional feedback* values. (*Channel 0 is unused in this mode.*)

[See 0x28MM - data for Analog Channels 4-7.]

It also contains indication that the byte-pipe mechanism has reset itself since the last AN03 packet was sent (Since the last configure sequence occurred.) This indication is in the form of two counters, one for the input pipe-half and one for the output pipe-half. Each of these counters can have a value of zero thru 15 where 15 means that 15 or more resets have occurred since the pipe counters were last reset.

The 0x24MM Module Status AN03 response packet consists of 8 bytes. Unused bits will be zero and should be ignored. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|--|
| 0.7-4 | * PipeHlthA: Count of IN-side timeouts [0-14, 15] |
| 0.3-0 | * PipeHlthB: Count of OUT-side timeouts [0-14, 15] |
| 1.7-4 | * PipeHlthM: Count of TxBfrFull events [0-14,15] |
| 1.3-0 | * PipeHlthF: Count of Pipe-writes received w/pwr off [0-14, 15] |
| 2.7-0 | Analog Channel 1 bits 7..0 |
| 3.7 | * PipeHlthG: FLAG=1 if pipe-IN busy, 0 if not |
| 3.6 | * PipeHlthH: FLAG=1 if pipe-IN in error recovery, 0 if not |
| 3.5 | * PipeHlthN: FLAG=1 if TxA/TxB bfrs both full, 0 if not |
| 3.4-2 | reserved (0's) |
| 3.1-0 | Analog Channel 1 bits 9..8 |
| 4.7-0 | Analog Channel 2 bits 7..0 |
| 5.7 | * PipeHlthJ: Flag=1 if pipe-OUT busy, 0 if not |
| 5.6 | * PipeHlthK: Flag=1 if pipe-OUT in error recovery, 0 if not |
| 5.5 | * PipeHlthC: Flag=1 Reset requested while pipe busy, 0 if not |
| 5.4 | * PipeHlthD: Flag=1 if writing packet to module, 0 if not |
| 5.3 | * PipeHlthE: Flag=1 if rcvd pkt but still writing prior, 0 not |
| 5.2 | * Module Switched Power State (1=on, 0=off) |
| 5.1-0 | Analog Channel 2 bits 9..8 |
| 6.7-0 | Analog Channel 3 bits 7..0 |
| 7.7 | * PipeHlthL: Flag=1 if Module I/F not ready, 0 if ready NOTE: E:IN-busy/J:OUT-busy or F:IN-errRcvry/K:OUT-errRcvry Describe side of I/F which is not ready and why |
| 7.6 | * CHECKSUM FLASH: 1=valid, 0=invalid |
| 7.5 | * CHECKSUM ERAM: 1=valid, 0=invalid |
| 7.4 | * CHECKSUM RAM: 1=valid, 0=invalid |
| 7.3-2 | Mode bits (M1,M0) showing jumpered mode |
| 7.1-0 | Analog Channel 3 bits 9..8 |

NOTE: * denotes Widget state feedback to control software. (This status is available for use by control software for diagnostic reasons, positive control feedback, etc.)

NOTE2: Analog channel-0 is not present in this packet as the signal is actually used for pipe-handshake control, not as an analog sensor.

NOTE3: Twenty-five of these unused bits actually carry byte-pipe-health data (PipeHlthA-N) which describe byte-pipe state machine condition to the controlling software. This is entirely used to diagnose problems with the byte-pipe state machine during Widget firmware and control software (file transfer, firmware download) development/testing.

0x28MM Module Status AN47

[8-BYTE PAYLOAD]

This is the second of two packets sent in response to a Module Configure Command to the IHU by a Byte-pipe-mode Widget.

This packet contains the 10-bit values for Analog channels 5-7. (*Channel 4 is unused in this mode.*) [See 0x24MM - data for Analog Channels 0-3.]

The 0x28MM Module Status AN47 response packet consists of 8 bytes. Unused bits will be zero and should be ignored. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|----------------------------|
| 0.7-0 | reserved (0's) |
| 1.7-0 | reserved (0's) |
| 2.7-0 | Analog Channel 5 bits 7..0 |
| 3.7-2 | reserved (0's) |
| 3.1-0 | Analog Channel 5 bits 9..8 |
| 4.7-0 | Analog Channel 6 bits 7..0 |
| 5.7-2 | reserved (0's) |
| 5.1-0 | Analog Channel 6 bits 9..8 |
| 6.7-0 | Analog Channel 7 bits 7..0 |
| 7.7-2 | reserved (0's) |
| 7.1-0 | Analog Channel 7 bits 9..8 |

NOTE: Analog channel 4 is not present in this packet as the signal is actually used for pipe-handshake control, not as an analog sensor.

0x04MM Byte-pipe data write to Module

[8-BYTE PAYLOAD]

The Byte-pipe mode Data Write packet consists of 8 bytes. There are no unused bits. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | Output Byte 0 (first byte written to interface) |
| 1.7-0 | Output Byte 1 |
| 2.7-0 | Output Byte 2 |
| 3.7-0 | Output Byte 3 |
| 4.7-0 | Output Byte 4 |
| 5.7-0 | Output Byte 5 |
| 6.7-0 | Output Byte 6 |
| 7.7-0 | Output Byte 7 (last byte written) |

NOTE: This is a low priority bulk-data CAN packet.

NOTE2: the tail end of this packet may not be transferred to the Module if the Module takes too long accepting bytes (if the output-timer expires).

Diagnostic NOTE: Should this happen the count of Pipe-out timeouts is incremented. The new value of this count will be seen by the IHU in the next AN03 configure-response packet sent by the Widget.

APPENDIX C: CAN MESSAGE SPECS

No response to this message is sent.

0x2CMM Byte-pipe data read from Module

[8-BYTE PAYLOAD, (0-7 BYTES IF TIMEOUT)]

The Byte-pipe mode Data Read packet consists of 8 bytes. There are no unused bits. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | Input Byte 0 (first byte read from interface) |
| 1.7-0 | Input Byte 1 |
| 2.7-0 | Input Byte 2 |
| 3.7-0 | Input Byte 3 |
| 4.7-0 | Input Byte 4 |
| 5.7-0 | Input Byte 5 |
| 6.7-0 | Input Byte 6 |
| 7.7-0 | Input Byte 7 (last byte read) |

NOTE: This is a low priority bulk-data CAN packet.

NOTE2: this packet may be shorter than 8 bytes if the input-timer expired aborting the transfer.

Diagnostic NOTE: Should this happen the count of Pipe-in timeouts is incremented. The new value of this count will be seen by the IHU in the next AN03 configure-response packet sent by the Widget.

0x14MM Module Census Query

[BROADCAST, 0-BYTE PAYLOAD]

NOTE: This message was intended to be used during Module development, and ground testing; however, there is no reason why the system developer could not use this in flight if deemed useful.

This message is used to poll a bus to see which Widgets are powered up/present. Since this is a broadcast, the address part of the 11-bit ID contains the sender's address, not the usual address of the intended receiver.

Widgets receiving this message should reply using their own address (in the address part of the message Id) with a 0x34MM response packet properly filled in. At first, this seems weird, but it prevents collisions on the CAN bus (all replying to same address) and provides for responses arriving to the requesting device in natural widget-priority order. The querying device simply listens for all 0x34MM packets irrespective of the "sending" address.

A device will send this packet then receive responses for a fixed amount of time thereafter (15 mSec should be sufficient for 63 devices on a bus.) After this time expires, it stops listening and assumes all powered-up devices have responded.

| Message Bits | Purpose |
|--------------|---|
| *** | There is NO message content (payload) for this packet *** |

0x34MM Module Census Response

[8-BYTE PAYLOAD]

A Widget constructs and sends one of these packets whenever it receives a 0x14MM Module Census Query packet. The widget must address this packet to its own address. The Querying device is listening for all 0x34MM packets ignoring the address field (the MM part).

The 0x34MM Module Census response packet consists of 8 bytes. Unused bits will be zero. The packet content is defined as follows:

| Message Bits | Purpose |
|--------------|---|
| 0.7-6 | Module Mode |
| 0.5-0 | Module Address |
| 1.7-0 | reserved (DEBUG RAM Checksum MsByte DEBUG) |
| 2.7-0 | reserved (DEBUG RAM Checksum LsByte DEBUG) |
| 3.7-0 | reserved (DEBUG ERAM Checksum MsByte DEBUG) |
| 4.7-0 | reserved (DEBUG ERAM Checksum LsByte DEBUG) |
| 5.7-0 | reserved (DEBUG EEPROM Checksum MsByte DEBUG) |
| 6.7-0 | reserved (DEBUG EEPROM Checksum LsByte DEBUG) |
| 7.7-4 | Widget Firmware Version (Major revision nbr) |
| 7.3-0 | Widget Firmware Version (Minor revision nbr) |

Diagnostic Messages (All Modes)

Messages in this section are used during firmware development and might not be flown in final flight firmware. They are included here for completeness.

0x18MM Module Read Counters

[3-BYTE PAYLOAD]

This message is used to retrieve health and status counters and diagnostic information from a CAN Widget. This is intended for use by control device while on the ground during verification of CAN Widget behavior. A control device sends one of these packets to the CAN Widget under test. The Widget will respond with a 0x38MM Read Counters response packet followed by 1-3 0x30MM Device Info packets (only if requested) and finally followed by 1-7 0x70MM ERAM Dump packets (again, only if requested). Therefore, depending on the contents of the request packet the number of reply packets can vary from 1 to 11.

Under normal use, a 0x01, 0x00, 0x00 payload instructs the Widget to clear its counters immediately after reporting them. However, a 0x00, 0x00, 0x00 payload may be sent if it is desired that the counters not be cleared.

| Message Bits | Purpose |
|--------------|---|
| 0.7-4 | Reserved (0's) |
| 0.3 | IPage3: 1=Send Page 3 of Device Info, 0=don't |
| 0.2 | IPage2: 1=Send Page 2 of Device Info, 0=don't |
| 0.1 | IPage1: 1=Send Page 1 of Device Info, 0=don't |
| 0.0 | ClrCtrs: 1=Reset Counters after Read, 0=don't |
| 1.7-5 | EPages: Return (1-7) 6-byte Pages of ERAM, 0=don't |
| 1.4 | ClrERAM: 1=Reset WD Event Storage aft. Read, 0=don't |
| 1.3-2 | Reserved (0's) |
| 1.1-0 | ERAM-Addr (MS-Bits) ERAM Address: [0x000-0x3FA] |
| 2.7-0 | ERAM-Addr (LS-Bits) (this value ignored if EPages=0) |

NOTE: The ending address is 0x3FA instead of 0x3FF since 6 bytes are returned in each packet. The address 0x7FA returns the final 6 bytes 0x3FA thru 0x3FF of the memory region.

NOTE2: 2-bytes of starting address and 6-bytes of data are returned in each 0x70MM ERAM dump packet requested. In this way, each packet is fully self-describing (no order of packet arrival is imposed). This also provides ability for the control software to verify that all packets requested have arrived. Irrespective of all of this, the Widget constructs and sends all packets requested in sequence from lowest address to highest.

0x38MM Module Read Counters Response

[8-BYTE PAYLOAD]

A Widget constructs and sends one of these packets whenever it receives a 0x18MM Module Read Counters packet. The widget must retrieve the “clear” control info from the payload of the packet and either clear the counters, or not, according to the control info found. The clear occurs AFTER replying to this request.

The 0x38MM Module Read Counters Response packet consists of 8 bytes. Unused bits will be zero. The module uses the following bits in the packet.

APPENDIX C: CAN MESSAGE SPECS

| Message Bits | Purpose |
|--------------|---|
| 0.7-0 | CANTEC value (transmit error counter - CAN ctrlr) |
| 1.7-0 | CANREC value (receive error counter - CAN ctrlr) |
| 2.7-4 | Nbr packets rcvd w/CANSTCH:DLCW set since last clear |
| 2.3-0 | Nbr packets rcvd w/CANSTCH:BERR set since last clear |
| 3.7-4 | Nbr packets rcvd w/CANSTCH:SERR set since last clear |
| 3.3-0 | Nbr packets rcvd w/CANSTCH:CERR set since last clear |
| 4.7-4 | Nbr packets rcvd w/CANSTCH:FERR set since last clear |
| 4.3-0 | Nbr packets rcvd w/CANSTCH:AERR set since last clear |
| 5.7-4 | Nbr packets rcvd w/CANGSTA:BOFF set since last clear |
| 5.3-0 | Nbr packets rcvd w/CANGSTA:ERRP set since last clear |
| 6.7-4 | Nbr packets rcvd w/CANGIT:OVRBUF set since last clear |
| 6.3-0 | Nbr packets sent causing TxFull set since last clear |
| 7.7-0 | Nbr of WatchDog fires since last clear |

NOTE: see [0x18MM payload0 bit0], which is set to one to clear counters.

NOTE2: the four-bit counters are actually 8-bit counters. The values shown in this payload are capped at 4 bits so values of 0x00 to 0x0E are accurate. However, 0x0F indicates 15 or more events have happened.

0x30MM Device Info Response

[8-BYTE PAYLOAD]

A Widget constructs and sends one of these packets whenever it receives a 0x18MM Module Read Counters packet that requests one or more device info pages be sent. Up to three pages can be requested; therefore, up to three of these packets can be in the reply sequence. A field within the packet identifies which page is contained. If the request is for more than one page, the pages are sent in ascending order (1 then 2 then 3).

The 0x30MM Device Info Response packet consists of 8 bytes. Unused bytes will have a value of 0xF? where the '?' is the page number requested. The following fields are defined in the Page-1 packet:

| Message Bits | Purpose | <Page-1> |
|--------------|--|-----------------------------------|
| 0.7-4 | Page Number [0x1] | (this is page 1 of 3) |
| 0.3-0 | Page-Definition Version Number [1-15]; 1=1st Def'n | |
| 1.7-4 | 4bits: [1-12] | Month of Date Flashed (UTC) |
| 1.3-0 | MS-4bits: [1-31] | Day of Date Flashed (UTC) |
| 2.7 | LS-1bit: [1-31] | Day of Date Flashed (UTC) |
| 2.6-0 | 7bits: [0-127] | Year(-1900) of Date Flashed (UTC) |
| 3.7-0 | MSByte of firmware Checksum | |
| 4.7-0 | LSByte of firmware Checksum | |
| 5.7-4 | Schematic Major Version | |
| 5.3-0 | Schematic Minor Version | |
| 6.7-0 | Widget Id Nbr [1-108] | |
| 7.7-4 | Batch Number [0-15]; (0x1 for first 108 made) | |
| 7.3-0 | Page check digit (checksum of this 8-byte page) | |

NOTE: see [0x18MM payload0 bit1], which is set to one to request this page be sent from the Widget to the IHU.

NOTE2: device info page data is stored in the high-end of the FLASH memory (following all functional code) and is placed in this region by the CAN-Do flasher utility during Widget re-flashing.

APPENDIX C: CAN MESSAGE SPECS

NOTE2: Pages two and three are include for future expansion.

The following fields are defined in the Page-2 packet:

| Message Bits | Purpose | <Page-2> |
|--------------|--|--------------------------------|
| 0.7-4 | Page Number [0x2] | (this is page 2 of 3) |
| 0.3-0 | Page-Definition Version Number [1-15]; | 0=Empty Def'n |
| 1.7-0 | Reserved | (0xF2) |
| 2.7-0 | Reserved | (0xF2) |
| 3.7-0 | Reserved | (0xF2) |
| 4.7-0 | Reserved | (0xF2) |
| 5.7-0 | Reserved | (0xF2) |
| 6.7-0 | Reserved | (0xF2) |
| 7.7-4 | Reserved | (0xF) |
| 7.3-0 | Page check digit | (checksum of this 8-byte page) |

NOTE: see [0x18MM payload0 bit2], which is set to one to request this page be sent from the Widget to the IHU.

The following fields are defined in the Page-3 packet:

| Message Bits | Purpose | <Page-3> |
|--------------|--|--------------------------------|
| 0.7-4 | Page Number [0x3] | (this is page 3 of 3) |
| 0.3-0 | Page-Definition Version Number [1-15]; | 0=Empty Def'n |
| 1.7-0 | Reserved | (0xF3) |
| 2.7-0 | Reserved | (0xF3) |
| 3.7-0 | Reserved | (0xF3) |
| 4.7-0 | Reserved | (0xF3) |
| 5.7-0 | Reserved | (0xF3) |
| 6.7-0 | Reserved | (0xF3) |
| 7.7-4 | Reserved | (0xF) |
| 7.3-0 | Page check digit | (checksum of this 8-byte page) |

NOTE: see [0x18MM payload0 bit3], which is set to one to request this page be sent from the Widget to the IHU.

APPENDIX C: CAN MESSAGE SPECS

0x70MM ERAM Dump Response

[8-BYTE PAYLOAD]

A Widget constructs and sends 1-7 of these packets whenever it receives a 0x18MM Module Read Counters packet that contains a request to dump ERAM. The request contains the number (1-7) of packets to construct and send.

The Widget's Watchdog-Event-Log is stored in ERAM (the lowest 104 bytes) A bit in the 0x18MM Module Read Counters Request is used to request an emptying of this log (similar to clearing counters).

The 0x70MM ERAM Dump Response-packet consists of 8 bytes. Unused bits will be 0. The packet is defined as:

| Message Bits | Purpose | |
|--------------|---------------------|-----------------------|
| 0.7-2 | Reserved (0's) | |
| 0.1-0 | ERAM-Addr (MS-Bits) | Addr: [0x0000-0x03FA] |
| 1.7-0 | ERAM-Addr (LS-Bits) | |
| 2.7-0 | ERAM(Addr+0) Byte | |
| 3.7-0 | ERAM(Addr+1) Byte | |
| 4.7-0 | ERAM(Addr+2) Byte | |
| 5.7-0 | ERAM(Addr+3) Byte | |
| 6.7-0 | ERAM(Addr+4) Byte | |
| 7.7-0 | ERAM(Addr+5) Byte | |

NOTE: This is a low priority bulk-data CAN packet.

NOTE2: see [0x18MM payload[1,2]], which identifies the ERAM start address and number of 6-byte pages to be sent from the Widget to the IHU.

NOTE3: see [0x18MM payload1 bit4], which is set to one to clear the ERAM based Watchdog Event Log after all dump packets are sent from the Widget to the IHU.

0x1CMM Module Jump to ISP Loader (deprecated)

[2-BYTE PAYLOAD]

This message is used to command a specified CAN Widget to enter In System Programming (ISP) mode wherein it can receive new firmware. Only one CAN Widget will enter programming mode. The other CAN Widgets will effectively remove themselves from the CAN bus so as to not interfere with the following ISP traffic on the bus.

All CAN Widgets will disable power at their 40 pin Module connector and then reply by sending the 0x3cMM Module Jump to ISP Response message indicating that they are taking appropriate action.

The CAN Widget which is specifically addressed in the message id will (after removing Module power and replying) jump immediately into the ATMEL ISP boot loader code effectively reconfiguring the device for CAN ISP messages at 500Khz bit rate ready to respond to ATMEL CAN ISP traffic.

The remaining CAN Widgets which are NOT specifically addressed by the message id will (after removing Module power and replying) simply shut down to be awakened again only by removal and re-application of power to the devices via the D15P CAN bus connector.

This mechanism allows a single command to be sent which will effectively reconfigure the CAN bus to appear as a single CAN Widget running in ATMEL CAN ISP mode. Therefore, we no longer have to re-cable the CAN Widgets in order to perform in-system programming. We simply send the command, reprogram the device, verify the device, and then remove and reapply power to the CAN bus resetting all CAN Widgets back into normal flight mode operation at 800Khz bit rate.

| Message Bits | Purpose |
|--------------|------------------|
| 0.7-0 | Fixed Value 0xF0 |
| 1.7-0 | Fixed Value 0x0D |

NOTE: The entire purpose of this payload is to decrease the likelihood that this packet can be received accidentally. With this payload, we have added 16 more bits to this packet, which must contain this fixed value.

0x3cMM Module Jump to ISP Loader ACK (deprecated)

[1-BYTE PAYLOAD]

A CAN Widget constructs and sends one of these packets whenever it receives a 0x1CMM Module Jump to ISP Loader packet. Just before sending this reply message, it removes power from the 40-pin Module connector.

A CAN Widget which receives the 0x1CMM packet and whose address appears in the message id will then jump into the ATMEL CAN ISP firmware. (It will signal this by setting byte0 of the payload to 0x01) A CAN Widget which receives the 0x1CMM packet but is not addressed by it will go into passive mode on the bus by disabling the CAN Controller and halting. (It will signal this by setting byte0 of the payload to 0x00)

| Message Bits | Purpose |
|--------------|--|
| 0.7-1 | Reserved (0's) |
| 0.0 | Action: 0=HALTing, 1=JUMPing to ISP Bootloader |



D: Control Software Development

The majority of this document presents information for our Module developers. In this appendix, we cover expectations the Widget development team has of the control side of the system in the form of a reading path for the Control Software (IHU) developer.

NOTE concepts in this section are for reference only. They may be more appropriate to the Eagle team and how they think about the issues or these concepts may only be viewed this way by the CAN-Do developers and may not be how the IHU firmware developers need to actually implement. In any case, this section describes detail and concepts, which inform as to how the CAN-Do team views the system operation. They should be useful to understand. Your mileage, however, may vary.

NOTE-2: this section refers to a configure frequency of 50Hz (20mSec.) which may or may not be used in your satellite. Please understand this to be the timing specified by your Satellite design team which may be different.

Reading Plan for the Control-software Developer

Welcome Control-software Developer to the world of Widget network control! In this section, we will help identify what you need to learn from this User's Guide which will help you effectively control a constellation of CAN Widgets and provide for diagnosis of problems within this constellation. As we are identifying what you need to know, we will also be identifying where to find specific related information in this users guide and in some cases at locations on the worldwide web. Let us get started.

As a control-software developer, you may need to:

- Create the basic transport system for control and telemetry data to and from all Modules on the spacecraft (hereafter called the constellation of Modules).
- Add bulk data transport support for any microprocessor controlled Modules. This support provides an ability to re-program firmware of a module and/or upload/download any bulk data (such as pictures from a camera module, etc.)
- Add lightweight diagnostic support for determination of the health of the constellation of widgets. This consists of counting transient error/warning occurrences in the CAN controller within the IHU and providing for retrieval of these counts and similar counts found in each of the Widgets in the constellation.
- Add application elements that control each of the specific Modules/Payloads in the constellation.

Please review the following sections of this User's Guide or CAN-Do Project website to obtain information needed to accomplish these tasks:

1. Read the section entitled "Error! Reference source not found." to obtain a conceptual overview of areas in support of your efforts.
2. Skim Appendix "C: CAN Message Specifications" in order to become familiar with the range of CAN packets with which you will be working.
3. Skim chapter "4: Firmware Overview" in order to better understand the widget itself and its basic capabilities.
4. Skim the three mode specific chapters to learn of why each mode exists and to gain a basic understanding of how the modes differ in function and offering to the Module designer.
5. If you have (or plan for any) Byte-pipe Mode Modules on your spacecraft then you will need to read and implement the **Can-Do Byte-pipe File-transfer Protocol (CDFTP)** within your control software. *You can download the latest CDFTP specification document from our CAN-Do project web site. (<http://can-do.moraco.info/doc/Byte-pipe-fileXfer-v04.pdf> at the time of this writing.)*
6. The CAN controller chip in your controller is likely different from that in the Widgets. You will need the detailed documentation for it and will need to determine which transient errors/warnings are available from this controller. You will then want add counters of these events, as they are lost otherwise as further packets are sent or received.
7. Lastly, you will be creating software that interacts with all modules on the spacecraft. You should expect to receive detailed CAN message specification for each Module which describes which bits control what in each module and which bits indicate which statuses along with the specification of the meaning of each of the Module defined analog channels.

As you are reading sections of this User's Guide, you will see there is much more information here than this reading plan suggests. Not all of this information may be useful to your task but you may find a read through this additional material may help you understand more of how and why the Widget is what it is today.

Control Software Concepts

This section introduces you the **Control-software Developer** to function needed within the control software. As we designed our Widget and its operating modes, we have come to expect certain behaviors from the IHU. While we are not writing the IHU code we have been discussing its behavior in order to come to how the Widget should work and what it can expect from the system. In this section, we present concepts we have been discussing so that you can understand the expectations this Widget system as implemented now has of the control software.

These next sections present a general view. The IPS-specific section (last section) of this appendix is written for IHU Firmware implementer consideration.

CAN vs. I/O Multiplexer and CAN Errors

The CAN bus and the Widgets along with the CAN controller in the IHU are replacing the I/O Multiplexer we have used in past satellites. We simplify interaction with the Modules by simulating this I/O multiplexer within the controller. Instead of regions of address space identifying endpoints of the I/O multiplexer, we

now simply allocate memory (address space) which still looks like these endpoints. However, in this new generation of control we layout the bits associated with each module in the CAN packet format. This memory now contains the outbound configure packets and the inbound response packets for all Modules on the CAN bus. The 20 millisecond interrupt handler then iterates over the set of configure packet images and sends each in turn over the CAN bus. The CAN receive interrupt handler then simply deposits each response packet it receives into the proper storage location dictated by the sending widget ID and the message type. This is all there needs to be to transport all control values to the Modules and to receive all telemetry from the Modules. Now we can think of this in memory image of all sent and received data as the ends of the wires connecting to the modules. Therefore, if an application within the control software wants to set an output value of one of the modules it simply writes to the location corresponding to the output bit designated for that Module. At the next 20-millisecond interrupt the new value will be sent to the Widget attached to that module. Any change in Module telemetry induced by the control change will then arrive and be posted to the corresponding telemetry location in memory during the remainder of the same 20-millisecond interval. If the change takes longer than the remaining portion of the 20-millisecond interval then new telemetry values will simply arrive in memory on the next 20-millisecond poll after the change occurs at the Module. Likewise, if the application wanted to read some telemetry from a Module is simply reads the memory location for that Module as that location always contains the most recent values returned from the Module. This new interrupt-based transport layer completely handles all normal traffic to and from all Modules on the Spacecraft.

Now that we are using a CAN bus to communicate we have imperfections in this data transfer effort to which we should pay attention. CAN controllers detect many error types during transmission and reception but the indication of these errors is only valid during or at the end of each packet. In order for us to make use of this “briefly known” error data we need to hold on to this information. The easiest (least amount of code, traditionally) means to do this is to have counters for these events that increment every time one of these events happens. We also need to be able to clear this set of counters. Each of the widgets has a form of this event counting which is as rich as the information the controller within the Widget provides. Together with the counts in the IHU we now have error-rate information for every device on our CAN cable. As we are exploring segmentation of the CAN cable to mitigate whole cable failures we now also have clear indications of which area of the cable is experiencing which type of problems. This can clearly help diagnose in-flight/testing transmission quality issues with our CAN bus.

Bulk Data – to/from Modules

If you find that imaging devices or other microprocessor controlled Modules are being flown which need firmware updates in flight then you will need to implement the CAN Do Byte-pipe File-transfer Protocol (CDFTP). This is the means you will use to send bulk data (firmware) to or receive bulk data (images) from the Module without affecting the delivery rates of control and telemetry data over the CAN bus. The CDFTP ensures accurate delivery of the firmware or image data while using the low-priority bulk data transport messaging capability of the AMSAT CAN message set implemented within our Widget.

Widget recovery

The IHU can request that a Widget turn on or off power to its attached Module electronics but there is no like control for affecting the power of the Widgets themselves. They are always powered on. However, if a Widget appears to be acting up the control software does have one control. If no configure messages are sent to a specific Widget for longer than 3.12 seconds then the Watchdog timer onboard the Widget will fire and cause the Widget to reset. You might think this would interfere with the Module electronics or function but each Module is designed so that it can lose power during this reset without affect and the Widget is designed so that the prior control values presented to the Module electronics will be quickly restored to the Module immediately following this reset.

IPS-centric discussion of Widget constellation control by Bdale, KB0G

The general idea behind the IPS side of the CAN-Do implementation is that we want to abstract the widgets into a pair of memory structures on the IHU-3, one representing the desired output state of all the widgets on the bus, and the other representing the current input state of all the widgets on the bus.

In addition to these in-memory state representations, we also need three packet queues, two on the transmit side and one on the receive side. The two transmit queues contain output packets for the command protocol, and output packets for the user data protocol(s), respectively. The use of two queues is the cheap hack for ensuring the command protocol always has priority, see below for more details. The single receive side queue is for user data protocol(s), and holds received packets until they can be processed by the appropriate IPS application task.

IPS words get defined to allow IPS application tasks to set output state elements and read input state elements without needing specific knowledge of the in-memory structure.

A task exists on the 20ms chain that reads the desired output state memory structure and queues a set of packets, one per widget on the bus, to set the output state on the physical hardware matching the desired output state in the memory structure. It is possible to conceive of only doing some widgets on each iteration, but a design objective for CAN-Do was to be able to refresh the widgets on every 20ms increment to minimize output state change latency and to ensure that any module output bits that get flipped for radiation or other unknown reasons are reset to the desired state quickly.

An interrupt service handler efficiently does most of the exciting work. There are two parts to this handler, one to handle pending transmit state changes and the other to handle pending receive state changes. Both should get a chance to run on each interrupt (each is a simple status bit check in the degenerate case).

When the CAN controller is able to accept more transmit packets, the transmit side of the interrupt handler empties the command transmit queue first and then processes packets pending in the user data transmit queue. Because packets sent by the IHU are always higher in priority than any other traffic on the bus, it should always be possible to empty the command transmit queue before the next 20ms tick re-invokes the above-described task.

When the CAN controller has received packets available, any that contain input state vector information are immediately copied into the right place in the input memory structure. Received packets that are part of the user data protocol are added to the received packet queue to be handled by IPS application tasks. It is completely reasonable for us to bound the size of this queue based on the number of byte-pipe widgets on the bus and our expectation of how many packets they are likely to be able to generate in 20ms.

Additionally, a low-priority IPS user task should get run from time to time to exercise the "out of band" capabilities implemented in the widgets to extract census information, record the current state of error counters, etc. This task queues request packets into the user data transmit queue, and registers as the interested listener for relevant reply packets in the receive queue. This task can make summary information and/or noteworthy error events visible for inclusion in routine spacecraft telemetry on the downlink.



E: Revision History

24 OCTOBER, 2007

Made general edits identified by early reviewers
 Revised Reading Plan and new Concepts Sections for Module Designers
 Added Reading Plan (appendix E) for Control Software Developers

17 JULY, 2007

Added Reading Plan and new Concepts Section for Module Designers
 Upgraded the Firmware Introduction section
 Updated byte pipe narrative
 Corrected Fig. 8 – 10V Standard Mode Example - Added R9 (between Q1 and Q2)
 Updated Appendix C – CAN Message Details as follows

- Updated all AN03 forms with new diagnostic bit fields
- Changed 0x18MM Module Read Counters from a 1-byte packet to 3-byte
this adds support to retrieve device info pages and ERAM WD Log content
- Add Byte 6 def'n to 0x38MM Module Read Counters Response
- Add 0x30MM Device Info Responses
- Add 0x70MM ERAM Dump Response
- Marked ISP messages as deprecated since they no longer worked after enabling the WD.

17 OCTOBER, 2004

Add new “**Firmware Overview**” Chapter
 Update byte-pipe chapter and add new “**Byte-pipe power-on stabilization interval**” section
 Update multiplex chapter with new dual-multiplex-pass behavior
 Update Appendix C – CAN Message Details (get ctrs: remove a control, jmp2isp: add payload)
 Incorporate version 1.6 Schematic

11 OCTOBER, 2004

Release third Version of User’s Guide
 Mark as specific to Firmware version 1.0 and earlier
 General text cleanup
 Fix table/figure broken references
 Apply changes suggested by our readers

22 NOVEMBER, 2003

Release second Draft of User’s Guide – correcting credits
 General text cleanup (more to do)
 Describe analog step size
 Add byte-pipe detail (more needed)

APPENDIX E: REVISION HISTORY

09 NOVEMBER, 2003

- Release first Draft of User's Guide (hardware and all modes)
- Adds Interfacing guidelines
- Adds example schematics for module interface logic
- Adds mechanical drawings
- Obsoletes the standalone mode documents
- Adds some detail for Byte-pipe mode
- Presents diagnostic CAN messages (supporting firmware development)

26 JULY, 2003

- Release 3rd Draft of Byte-pipe mode standalone document (Pin tasking change!)

30 JUNE 2003

- Release 2nd Draft of Standard mode standalone document
- Release 2nd Draft of Byte-pipe mode standalone document
- Release 3rd Draft of Multiplex mode standalone document

27 JUNE 2003

- Release 1st Draft of Standard mode standalone document

16 JUNE 2003

- Release 1st Draft of Byte-pipe mode standalone document
- Release 2nd Draft of Multiplex mode standalone document

05 JUNE 2003

- Release 1st Draft of Multiplex mode standalone document