Customizing CANopen for Use in an Automated Laboratory Instrument

M. B. Simmonds, Quantum Design Inc. and Olaf Pfeiffer, Embedded Systems Academy

We describe an optimization of the CAN physical layer as well as the CANopen application layer for use as an internal bus in a line of modularized laboratory instruments. Modifications and extensions are described for the pin assignments, Default Connection Set, Emergency Object, and the Device Profile to better support the requirements of our hardware. We also present a method that facilitates updating a module's firmware via CANopen.

Background

Our products are relatively complex cryogenic instruments used by physicists and chemists to perform research in material science. These instruments contain several GPIB (IEEE-488) modules that are controlled by the operator from an application running on a PC. The GPIB was chosen primarily because it was widely used by the scientific/engineering community at that time and enjoyed substantial hardware and software support. It also enabled users to integrate their own 3rd-party instruments into the measurement system.

As we begin looking toward a more modular and modern architectures for our products, however, the shortcomings of the GPIB are becoming more evident. The cost, complexity, and cable size for this 8-bit parallel bus becomes very unattractive when we contemplate using it with a larger number of modules. Even the stacked 26-pin ribbon connectors become a major size problem.

Furthermore, the protocols for required for exchanging short packets with an array of modules is very time-consuming and negates all the advantages one would expect from a parallel bus: we see an effective bit-rate for actual data of only about 200Kb in our systems.

For these reasons, we began searching for an alternative among the various serial busses that have become popular since our original decision was made almost two decades ago. We looked carefully at physical layers based on RS485, FireWire, EtherNet, USB, and CAN. We chose CAN because of several perceived benefits: non-critical cables and connector impedance requirements, good hardware support at the chip level, excellent bus arbitration and error checking, and adequate bandwidth. While we were initially impressed by the promise of very high bit-rates available in other busses, a closer evaluation showed that for our system we would be better off with the shorter frames and inherent collision-avoidance provided by CAN. Also, the high bit rates of these busses would limit our cable lengths or turn impedance matching into a serious design concern.

Having chosen CAN for the lower-level protocols, we needed to select (or invent) an "application layer" for our system. Several options were available, all based upon CAN: DeviceNet, CAN Kingdom, SDS, and CANopen. Here the decision became more a matter of taste since all of these approaches appeared to offer a reasonable set of features. The most important service we required (and did not want to reinvent) was a confirmed exchange of messages longer than 8 bytes: a Service Data Object in the terminology of CANopen. DeviceNet and CANopen appeared to be the most widely used and best supported of these options, with DeviceNet enjoying a much greater presence in the United States. But since it has never been our intention to market fieldbus devices for use except as internal components in our laboratory instruments, this bias toward DeviceNet was not a particular concern for us. This higher baud rate and more efficient block transfers offered by CANopen were of greater importance.

Lab Instrumentation Requirements vs. CAN Physical Layer Specification

The modules comprising our instrument require several electrical services in addition to CAN communication. We need ±24V power, 50/60Hz line synchronization, hardware reset, and a low-jitter hardware synch signal. We also want to provide separate paths for returning unbalanced supply currents, for establishing system ground reference, and for dumping shield currents. Table 1 shows how we adapted CAN's 9-pin D-sub connector to fill all of these requirements. Note that it is possible to connect a standard 3rd party CAN module into our network by using a cable with wires on only pins 2, 3, 7, and 9. In this case, the 24V supply would only provide power for the galvanically isolated CAN interface of the module. We are not using galvanic isolation of our CAN interfaces, so the ±24V supplies all power requirements in our modules.

The 50/60Hz sync line allows us to make very stable measurements in the presence of substantial line interference. Non-synchronous measurements are prone to exhibit low-frequency beats as their phase slowly slips with respect to the power lines.

The SYNC-H/RS and SYNC-L lines allow us to distribute a very accurate and stable timing signal throughout the system. This differential signal can serve as a clock, sync, or trigger for various modules depending on their requirements. The sub-microsecond latency and jitter available through this SYNC mechanism is far better than we could have obtained through the CAN bus itself. Commands sent over the CAN interface can be used to configure or arm modules so that they can make use of this timing signal as desired.

We will use CAN transceiver chips to control these SYNC lines, so in normal operation they will have the same electrical characteristics as the CAN bus. However, pulling Sync-H/RS to system ground level for a few

Pin #	CAN Standard Pinout	QD-CAN Pinout
1	Reserved	-24VDC Supply
6	Optional Ground	System Ground
2	CAN-L Line	CAN-L Line
7	CAN-H Line	CAN-H Line
3	CAN Ground	24VDC Return
8	Reserved	SYNC-H/RS Line
4	Reserved	SYNC-L Line
9	CAN_V+ Optional Supply	+24VDC Supply
5	CAN_SHLD Optional	Line-Sync (50/60 Hz)

Table 1: Comparison of pin assignments on D-Sub Connector

microseconds will initiate a hardware reset of all modules connected to the bus.

Lab Instrumentation Requirements vs. CANopen Specification

As already mentioned, we were selecting a serial bus for internal use in our instrument lines, therefore slavish adherence to an official specification was not required. Nevertheless, we wished to avoid "reinventing the wheel" to as great an extent as possible. Our earlier designs had also suffered from incompletely engineered and under-documented interfaces between the felt that we could reduce such problems by following an official standard that many people had already spent considerable time designing.

Also, we want to maintain the ability to components of our instruments. It was run 3rd party CANopen modules on our instrument's bus. Therefore, any liberties we choose to take with the DS-301 specification must be compatible with this requirement. The converse is not true, however: we do not care that our own instruments do not function correctly in someone else's network or if our instruments fail to pass CIA conformance testing.

CANopen fieldbus system and the bus required for our instruments. These differences are summarized in Figure 2. As one can see, the developers of CANopen were attempting to solve a very different set of problems than we are. Nevertheless, the CANopen application layer comes fairly close to providing our company with the necessary and sufficient services we require.

Our modules are quite application specific and can be pre-configured to perform their assigned functions in our instruments. There is no need to have dynamic assignment of PDO data nor is there even a need to have configurable COB-IDs for the PDOs. In fact it is desirable to have all these parameters "hard-wired" into the firmware so that our modules know everything about each other at power-on. Because of this determinism, no LMT and DBT capabilities are required on our bus.

Optimizing the Default Connection Set

Since we are adopting a fully static configuration of PDOs, the "default connection set" for our network must provide maximum capability and make it possible for modules to send as much process data as they may need to. The CANopen specification only allows for 4

Typical CANopen Fieldbus	Laboratory Instrumentation Bus
Every implementation quite different	Most instruments basically identical
Large number of simple modules	A few complex modules
Several interchangeable vendors	Vendor makes, uses own modules
Only a few generic module types	Unique, application-specific modules
Substantial module configuration req'd	Modules wake up knowing their role
Modules exchange process data	Users' computer collects process data
Minimal SDO traffic when "operational"	Commands continually sent via SDO
Computer used for config & diagnostics	User runs instrument through computer

Figure 2. Differences between typical CANopen System and Lab Instrument bus

There is a substantial difference between the "flavor" of a typical

TxPDOs and 4 RxPDOs per node, a number that we felt was insufficient for our system requirements. On the other hand, the number of nodes permitted by the CANopen specification was far in excess of what would be needed for our instruments.

We have therefore decided to make a tradeoff: limit the nodes to 31 in order to expand the number of default TxPDOs available on each node. Since our modules will serve primarily to control the instrument and report back process data, it is the TxPDOs, as opposed to the RxPDOs that are in short supply. We have therefore devised a strategy for "stealing" COB-IDs of the default PDOs we are excluding from our instrument network (32-127).

The technique is to allow each node in the range 1-31 to have three additional images in the range of 32-127. Thus, node #1 also inherits the default PDOs for nodes #33, #65, and #97. The COB-IDs for both RxPDOs and TxPDOs in this range are taken for use as TxPDOs for our modules. We also have available to us the COB-IDs of the default SDOs for these unused nodes. We thus have a total of 34 separate Process Data Objects available on each module for reporting data back to the user's computer. Note that we have retained the four (4) RxPDOs provided by the CANopen standard as part of our own default connection set. The order for assigning COB-IDs to these 34 PDOs is shown in Figure 3, and was chosen so that they would be used in order of decreasing priority.

Since we are not allowing the COB-IDs to be changed, the values listed in Figure 3 can be relied upon: the control computer and the other nodes automatically know a PDO's source node and number from its COB-ID. And since dynamic data mapping is not allowed in our network, the type and meaning of the data payload is also immediately known throughout the network.

	Std. CANopen	**QD-CANopen**
Maximum nodes in system	127	31
Default TxPDOs / node	4	34
Default RxPDOs / node	4	4
Default SDOs / node	1	1
Baud rates	10-1000Kb	500, 1000Kb
Dynamic PDO Mapping	Optional	No
Variable COB-IDs	Optional	No
Remote Response	Optional	No
29-Bit Identifiers	Optional	No
LMT Services	Optional	No
SDO Block Transfers	Optional	Mandatory
Error Control Protocol	Guarding or Heartbeat	Heartbeat
±24V System Power on Bus	No	Yes
Sync/Reset Signals on Bus	No	Yes
Line-sync Signal on Bus	No	Yes
Compatible with DS-301 Net	Yes	No
Compatible with QD-CANopen	Yes	Yes

Figure 3. Comparison of standard CANopen and QD-CANopen

Although we are not allowing the COB-IDs to be changed, we do allow bit #31 in the dictionary entry for PDO communication parameter/COB-ID to be set or cleared. According to DS-301, setting of this bit invalidates the PDO and may prove useful in managing bus bandwidth with so many default TxPDOs potentially defined.

Figure 4 summarizes the differences we have described so far between the CANopen standard and our own adaptation of it.

QD TPDO	Default Connections	Assigned COB-ID
1	TxPDO #1 on N	0x180 + N
2	TxPDO #1 on N + 32	0x1A0 + N
3	TxPDO #1 on N + 64	0x1C0 + N
4	TxPDO #1 on N + 96	0x1E0 + N
5	RxPDO #1 on N + 32	0x220 + N
6	RxPDO #1 on N + 64	0x240 + N
7	RxPDO #1 on N + 96	0x260 + N
8	TxPDO #2 on N	0x280 + N
9	TxPDO #2 on N + 32	0x2A0 + N
10	TxPDO #2 on N + 64	0x2C0 + N
11	TxPDO #2 on N + 96	0x2E0 + N
12	RxPDO#2 on N + 32	0x320 + N
13	RxPDO#2 on N + 64	0x340 + N
---	---	---
29	TxSDO on N + 32	0x5A0 + N
30	TxSDO on N + 64	0x5C0 + N
31	TxSDO on N + 96	0x5E0 + N
32	RxSDO on N + 32	0x620 + N
33	RxSDO on N + 64	0x640 + N
34	RxSDO on N + 96	0x660 + N

Figure 4: QD Connection Set TxPDOs on Node N (0 < N < 32)

Enhancing the Role of the CANopen Emergency Object

Specification DS301 appears to leave quite a bit of flexibility in the use of the Emergency Object for device-specific purposes. There are several blocks of Error Codes that have been provided to facilitate this: F0xxh is for “Additional Functions”, FFxxh is covers “Device Specific” errors, 50xxh covers “Device Hardware” errors, and the entire “6xxxh” block is available for Device Software errors.

We are extending the definition of “emergency” to include any significant events or state changes that might occur in a module, but whose actual occurrence would not otherwise be known without performing continuous polling of the module. Having to do such polling is a substantial programming burden and adds unnecessarily to the loading on the CAN bus. Also, such polling cannot be done by another node on the network unless it has Client SDO capability: a service not supported by some commercial CANopen slave stacks.

We propose to use the block of codes from F000h to FFFFh indicate that there has been a change-of-state in one of the modules subsystems. One bit (of the available 12) is assigned to each subsystem that can have externally significant state information. Whenever there is an event or state change in one of the module’s subsystems, the corresponding bit-flag in the Error Code is set. We provide an entry in the Object Dictionary for the purpose of clearing the flag-bits of this Error Code: the “Event Reset Register”. Setting a bit of this object clears the corresponding flag of the Error Code. According to the Emergency Object specification described in DS301, the EMCY telegram is sent when (and only when) the Error Code changes. Thus clearing any bits in the Error Code will cause the EMCY telegram to be sent again. But rather than sending an Error Code of 0x0000 upon resetting one of these bits (as mentioned in the standard), we propose to send the new Fxxxh pattern. Clearing a bit in the Fxxxh group indicates that the module has been re-armed to send an EMCY telegram when another state change occurs on that subsystem. Otherwise no further state changes will be announced. We will institute a suitable EMCY ‘holdoff time” in order to avoid consuming excessive bandwidth through this module-state reporting scheme.

The five (5) bytes of the “Manufacturer Specific Error Field” provide a set of status flags and mode bit-fields. Up to 40 bits of state/mode information can be communicated with this scheme.

There has been considerable discussion about “borrowing” the official CANopen emergency protocol for the posting of state-change information. The alternative would be to implement the above scheme using PDOs. We have selected to use the emergency protocol for several reasons: it gives these messages a higher priority than all normal PDOs, it allows state information to be presented by a module even when that module is in the Pre-operational or Stopped mode, and it conserves COB-IDs. In the case of our particular CANopen master API, emergency messages have their own dedicated queue and callback function. This should make them somewhat less likely to become lost.

Providing for Application Firmware Update Via CANopen

We need the capability to update a device’s firmware by loading new executable code directly through the devices own CAN interface. This requirement creates an interesting challenge for the firmware architect since the CANopen stack is an integral part of the application firmware and must be compiled together with it. We have decided that the most reliable and robust method for implementing this capability is to have a “CANopen Loader” permanently available on the module. This minimal operating system only needs to provide a few services. It must be able to implement an SDO-Server download, it must be able to verify the checksum of the program it has downloaded, and it must be able to transfer command to the downloaded program. Once the new downloaded program initializes and begins execution; it completely replaces the loader and provides the code to implement a CANopen interface for any further communications.

We have two separate banks of Flash memory available on each module. One bank contains the CAN loader firmware in a write-protected area segment. The other bank is available for storing downloaded application code. When the device is first powered-up or after a hardware reset, program execution transfers to the loader program. The loader can verify that the currently stored application has the correct checksum as part of its initialization process.

When the loader starts, the node is in a special state not described within the CANopen specification. Entry into this mode is signaled by a Bootup Message with a node number that is offset from the actual node by a value of 0x20 (0x720+NodeID). This would normally not be a valid Bootup Message within our restricted pre-defined connection set where we only allow a range of NodeID’s (1-31), so it can be interpreted as an entry into the “System State”. It can receive data and report status via SDO, but it has no access to the application’s Object Dictionary and cannot process any PDOs. If the checksum of the current application firmware is determined to be correct by the system code, the node can be sent into its normal “Preoperational” mode by sending the usual network command. Alternatively, new firmware can be downloaded by use of SDO writes. After the new firmware has been loaded, execution can be transferred over to it by bank-switching between the two memory blocks. After initialization, the “real” application sends a standard Bootup telegram and enters its

Preoperational mode. By using bank switching we avoid having to re-map the interrupt vector table: a new table is automatically loaded in the operation.

Creating a Manufacturer's Device Profile

Our modules are not intended for use on CANopen networks apart from our own internal instrument bus. Therefore we are free to create our own device profile with a common set of dictionary entries in the range 0x6000 – 0x9FFF. According to the specification, non-standard device profiles should be indicated by a Device Profile Number of zero in the Device Type entry (0x1000) of the devices Communication Profile. The 16 high bits of this entry are available to specify "Additional Information". We will place a characteristic version number in this location so that our system software can distinguish between different revisions of our device profile.

Our device profile will provide a device-independent structure for accessing common information such as module temperatures, module voltages, firmware checksums, error registers, and diagnostic test results. SDO writes to a standardized dictionary entry will be used to command various levels of diagnostic tests.

Conclusions

Completely standardized CANopen would come remarkably close to filling the needs for our modularized instrumentation. We will use PDOs to report measurement results back to the master node in the PC. SDOs will be used to set or read parameters as well as to issue confirmed commands to the nodes. Our modification of the Default Connection Set, our expanded scope of the Emergency Object, our provision for CAN-Based firmware updates, and our customization of the Device Profile go a long way toward making this high-level protocol a perfect fit to our requirements.

Quantum Design
11578 Sorrento Valley Road
San Diego, CA 92121
www.qdusa.com

Michael B. Simmonds
P: (760) 926-8673
info@qdusa.com

Embedded Systems Academy
50 Airport Parkway
San Jose, CA 95110
www.esacademy.com

Olaf Pfeiffer
P: (408) 910-7899
info@esacademy.com