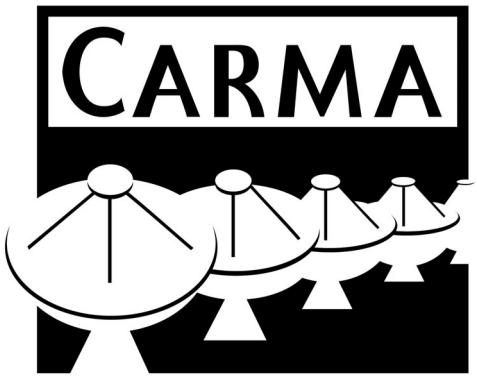

Combined Array for Research
in Millimeter-Wave Astronomy

GeneriCAN: CAN Node Design for CARMA

James Lamb, Jim Fredsti, and Brad Wiitala

1.0

February 6, 2003

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		

GeneriCAN: CAN Node Design for CARMA

James Lamb, Jim Fredsti, and Brad Wiitala

Abstract

We present the basic design of hardware and software for a generic CAN node based on the experience at OVRO. The intention is not to present a single CAN node design to perform all functions, but rather to provide a template from which to develop hardware and software for specific applications. A certain minimum hardware functionality is provided, with a prescription for adding new capabilities. A kernel of software will be provided to implement core capability such as scheduling of tasks in a time-critical manner, driver software for various types of devices, and templates on which new projects can be based.

Many CARMA-specific requirements are already implemented. These include: precision timing and time stamps; a CAN connector definition that includes *Reset*, *Timing*, and *Power* lines; a CAN address format standard; and fast and slow monitor package handling.

Numerous different CAN nodes will be required for CARMA so it is important to maintain reliable control of code and documentation. No specific procedures are currently implemented, but these must be implemented as a CARMA-wide solution.

1 Introduction

Embedded processors communicating using the CAN protocol will be widely used in CARMA. All CAN nodes that are to be integrated into the CARMA system have to interact with the higher level software in a homogeneous way, accepting commands with standard addressing and formatting, and streaming back monitor information with a predefined format and timing. While nodes for different applications will require different hardware and software implementations, there will be a core of functions that are similar across virtually all the implementations. There will also be a pool of hardware and software fragments that can be used as a 'mix-and-match' resource for new designs

In this document we outline a basic hardware design and core set of software based on developments at OVRO over the last two years. Modules, such as the COBRA downconverter controllers, have been implemented and successfully integrated into the OVRO control system. It is proposed that this 'template' design will be reviewed for general CARMA suitability and a core set of hardware and adopted for all new designs. Note that his does not imply that all CAN nodes are identical, but just that common functionality is uniformly implemented.

A CAN node has an embedded microprocessor controlling one or many hardware functions of devices that are located close to the node. Each node should be recognized by the high-level system when it is plugged into an appropriate bus. Messages on a CANbus have headers that

identify devices and control priority. A standard for messaging has been adopted by OVRO [1]. The 29-bit Message ID allows the CAN node to be addressed either by API or Board Type and whether or not the Linux host should ignore the message. Message types (with several standard ones defined) and board or API type are also included in the ID. Data may be included in payloads of up to eight bytes as required. This standard appears equally suitable for CARMA use.

CAN hardware node type and version information is stored in unique ID 1-Wire® devices, and the API designation visible to the high-level system type is embedded in the software. A single hardware design may be programmed with several API types, and a single API type may be implemented on different hardware implementations (mainly hardware revisions that do not change the functionality presented to the high-level software).

2 Hardware Description

2.1 Overview

The CAN hardware is based on a commercial module (a phyCORE "single board computer") that plugs into a host board designed to implement whatever hardware functions are required for a given application. While each application will probably require a different board design, there is a core of functionality that should be implemented on virtually all boards. This ensures some uniformity of parts and allows certain software modules to be re-used with no changes other than defining constant values. Circuit diagrams are available in OrCAD format for inclusion in future designs.

Because of the limited number of I/O ports on the phyCORE module, significant additional circuitry is required for attaching many peripheral devices. This should be accomplished with the same method of decoding for all applications so that the same software drivers can be used (differences in address for the same device on different boards can be accommodated by using a `#define` for the address in the project header). A digital I/O scheme is included in this design for this purpose.

The functions that have been implemented to date are described below, and the list will continue to expand.

2.2 Processor Module

The Phytec phyCORE XAC3 module [2] uses the PhilipsXAC3 microprocessor with in-built CAN communications. The module has two sets of pins that allow it to be plugged into the board that will hold all the application-specific circuitry. In addition to the processor the module includes: memory; address decoder; RS-233, RS-485, and CAN drivers; a real-time clock (RTC). The XAC3 chip is socketed, so it can be replaced by an in-circuit emulator (ICE) for debugging in real time.

While the phyCORE module has some shortcomings, particularly in the number of digital I/O lines, it has most of the functionality required for CARMA applications. The module board has components on both sides, and components may be placed underneath it on the host board, yielding a high component density in a 55 mm × 47 mm area.

2.3 Serial Port

The RS-232 serial port is available on a D-connector and is the primary means of downloading programs and debugging software.

2.4 CANbus Interface

CARMA ICD 6000 specifies the physical and electrical characteristics of the CANbus interface. In essence it specifies an RJ-45 connector that carries the CANbus signals, a *Reset* pair, a *Timing* pair and a *+12 V Supply* that can be used to power CAN nodes that have modest power requirements.

2.4.1 CAN bus

Drivers for the CANbus are incorporated on the phyCORE module. The high-speed bus (1 Mb/s) is implemented. Typically, a CAN node has two RJ-45 connectors so that nodes may be easily chained together, with the CAN drivers connected in a 'T'. Restrictions on the length of the 'T' are given in ICD 6000 and are easily met.

2.4.2 Reset

A remote reset is implemented that can be used to reset the microprocessor and any other hardware on the board. For noise immunity it is implemented as an RS-485 signal on a twisted pair as described in ICD 6000.

2.4.3 Timing

Timing signals are brought on to the board on the CAN connector. The timing signal (*e.g.*, 1 pps) must be the same for all nodes on a given bus, but different nets can have different timing signals as required.

2.5 Data and Address Lines

19 address lines and 16 data lines are available from the phyCORE module. Some of the address lines are used for digital I/O (see below).

2.6 Ports

Eleven ports are available from the phyCORE module. Almost all of these are allocated in the template design as follows:

Port	Function
1	SPI
2	SPI
3	SPI
4	SPI
5	1-Wire Bus
6	UART (RS-232/RS-485)
7	UART (RS-232/RS-485)
8	External Interrupt, Timing Reference
9	External Interrupt
10	Unallocated
11	Unallocated

2.7 Timers

There are three timers. One is used for the pacer (see below) and one for the UART, while the third one may be used as needed.

2.8 Real-Time Clock

Accuracy of the real time clock is not sufficient for most CARMA time-critical tasks. These are handled with a timer and the operating system software. Non-critical applications may use the RTC as appropriate.

2.9 Digital I/O

Potentially there are up 138 read and 138 write bits. Typically 8×8-bits are implemented. Of these, the first byte is allocated for software LED's, and one byte for chip select addressing. Both are described below.

2.10 Status LED's

Several LED's are used, three directly connected to the hardware and the remainder under software control. At least five of those under software control should have the same functions for all nodes. The remainder may be used for application-specific purposes, or not included on the board.

2.10.1 Hardware LED's

These are connected directly to the relevant hardware:

LED	Function	Color
1	Reset	Red
2	Timing	Yellow
3	5 V present	Green

2.10.2 Software LED's

Software LED's are connected to 1 byte of the digital outputs at. Their standard functions are:

LED	Function	Color
1	Pacer (5 Hz)	Green
2	CAN Error	Yellow
3	CAN Tx	Yellow
4	CAN Rx	Yellow
5	General Error	Red
6	Unallocated	–
7	Unallocated	–
8	Unallocated	–

2.11 One-Wire Bus and Devices

A general 1-Wire bus is implemented. All boards will have at least one 1-Wire device, a DS2430A, that is used to identify the board type and serial number in a standard format [1]. Many boards will have a second 1-Wire device that provides the location of the board. The second device may, for example, be on a backplane that the board is plugged into to indicate which slot it is inserted in. This allows the software and hardware to be completely generic for a given board type but allow the high-level software to distinguish among different units without resorting to board jumpers or software constants.

Other devices may be connected to the bus for temperature monitoring, remote ADC's, *etc.*

2.12 SPI Devices

Many devices use the Serial Peripheral Interface protocol. Although the hardware specification is the same for all, there are different implementations of the timing that are handled in the software driver. Multiple devices may be connected to the SPI bus. There are two different ways that can be used to address different devices. The simplest is to connect all the SPI devices to the same SPI bus and to select the device using a chip select (CS) line. The template design uses two bytes from the digital I/O, for a total of 16 devices. The second method is applicable only to certain devices that may be chained together. The data are clocked through all the devices and then strobed into them all at the same time.

Some of the devices that have been incorporated to date are as follows:

2.12.1 Memory

An 8k × 8 EEPROM (Atmel AT25640) is implemented for storing data. Although re-writable ROM is available on the phyCORE module, having persistent memory on the application board keeps the data with the board even if the phyCORE module needs to be changed.

2.12.2 ADC's and DAC's

The following analog I/O devices have been implemented:

Device	Function	Comments
AD7707	16-bit, 3-channel ADC	
MAX1270	12-bit, 8-channel ADC	
LTC1658	14-bit DAC	
LTC1668	16-bit DAC	
MAX3110	SPI UART	U. Chicago
AD7676	16-bit ADC	U.C. Berkeley
ADG528F	8-ch MUX	U.C. Berkeley
AD7841	14-bit DAC	U.C. Berkeley

2.12.3 Temperature Sensors

An AD7814 temperature sensor has been implemented, and is useful for monitoring the temperature of the module.

2.12.4 UART

A MAX3110 SPI UART (separate from the program/debug UART) that is required for controlling Zaber linear actuators for Gunn tuning and attenuator setting via an RS-232 control is being implemented by U. Chicago.

2.13 Voltage Monitoring

All power supply voltages on the board are monitored *via* the MAX1270 ADC.

3 Software Design

3.1 Overview

There is no standard operating system provided for the phyCORE module, so one has been developed for OVRO applications that should be suitable for CARMA use. Most of the programming is done in C, with some time-critical sections in Assembly Language. Code can be divided into three sections: the core operating system, drivers, and the application code.

The first should be regarded as 'read-only' and should be regarded as uniform across all applications. Releases will be as a package and have version number information embedded.

The core of the operating system is contained in four files, `opsys.c`, `pacer.c`, `errors.c`, and `canerrors.c`. Each device will have a driver in a separate file `<name>dvr.c`. These are all supplied as source code and contain a header file, `project.h`, that can contain defines for values to use in the system software, such as names for the status LED's.

Each implementation of a CAN node will have code specifically written for it. Most applications will require code to communicate with the serial port for debugging purposes. Although the commands and responses will vary greatly from one application to another the basic functionality will be similar. A code template, `monitor.c`, is therefore provided with some skeleton code that can be extended as appropriate. Other files will contain functions that can be sent to the task dispatcher for executing specific tasks.

Any given application will be built with all the standard operating system code, the drivers for the devices used, and the application code. The monitor code may be included or excluded from the final build, depending on speed and memory restrictions.

Most of the functionality is described in detail in [3, 4] and a summary is given below.

3.2 Operating System

3.2.1 opsys

`opsys.c` contains the core of the operating system for the Phytec module. It is responsible for managing the task queue and scheduling and dispatching tasks to the pacer. Input on the debugging serial port is detected and queued.

3.2.2 pacer

Functions in `pacer.c` take care of maintaining absolute time and timing of all tasks. They respond to the timer interrupt to start tasks on appropriate time boundaries.

3.2.3 errors

A general error handler with error logging is implemented in `errors.c`.

3.2.4 vectors

Definitions of the interrupt vectors used by the software, and hardware (timers, CANbus, UART, and SPI) are contained in `vectors.c`.

3.3 Drivers

Several drivers have already been implemented and more are in the process of being added. These include:

3.3.1 Bus Drivers

3.3.1.1 UART

`uartdvr.c` handles the I/O on the serial port, providing a FIFO buffer for data. It also allows CAN messages to be echoed to the serial port.

3.3.1.2 CAN

`candvrs.c` deals with receiving and transmitting CAN messages and handling errors on the CANbus. Only messages destined for the particular CAN node are accepted and put into a

buffer. A function is provided to generate a CAN address that is compliant with the OVRO-defined protocol.

3.3.1.3 1-Wire

The 1-Wire driver in `onewiredvr.c` will search for all devices on the 1-Wire bus. In particular it will identify the devices that hold the CAN node type and serial number, and the CAN node location.

3.3.1.4 SPI

Various possible hardware timing sequences for the SPI bus are coded in `spidvr.c`, with sections in Assembly Language to speed up communications.

3.3.2 SPI Devices

SPI drivers have been written for the following devices:

Device	Driver File	Function
AD7707	`ad7707dvr.c`	16-bit, 3-channel ADC
LTC1658	`dacdvrs.c`	14-bit DAC
LTC1668	`dacdvrs.c`	16-bit DAC
AT25640	`eeromdrv.c`	8k × 8-bit EEROM
MAX1270	`max1270dvr.c`	12-bit, 8-channel ADC

3.3.3 Other I/O

Routines for turning LED's on and off at specified intervals, internal RAM checks and various other routines are contained in `otherio.c`.

3.4 Application Layer

3.4.1 Application Code

The top level application code, `<projectname>.c`, contains the `main(void)` entry point to for the code. It supplies initialization routines, state machines, unit conversions, hardware control algorithms, *etc.*

3.4.2 Pacer Tasks

Any tasks that have to be dispatched to the pacer are defined in `pacertasks.c`.

3.4.3 CAN Messages

Many CAN messages will be the same for all CARMA nodes, and most nodes will have a set particular to the application. All messages, standard and application-specific, are coded in the file `canmsgs.c`. Code in this file is responsible for interpreting incoming CAN messages and activating the appropriate code in response. It also assembles and sends out the monitor packages. An important task of this code is also to take care of time synchronization requested by the CAN messages.

3.4.4 RAM/ROM

`ramromdefs.c` contains declarations of RAM and I/O address spaces.

3.5 Utility Code

3.5.1 Conversions

Sets of routines in `genrtn.c` handle conversions between different data representations (bytes, BCD, strings, *etc.*). There are also several utility functions for operations on strings.

3.5.2 Tables

At present all table operations are coded in the applications where they are used. In future a set of standard functions could be extracted into a single file to be compiled directly or for cut-and-paste into other source code.

3.5.3 Date/Time

Currently the software provides both coarse and fine module time control. Coarse time control guarantees better than 10 ms resolution while fine control guarantees better than 100 µs resolution. The fine control uses the *Timing* signal available as interrupt (0). The coarse timing uses time stamps sent from the system host which is synchronized to NTP.

4 High-Level Software

Although this document is primarily intended to cover the CAN node itself, it is useful to note that there has been concurrent development of high-level software for the CARMA-standard cPCI crate running Linux with a Janz CANbus card.

5 Documentation and Version Control

5.1 Hardware

At some point a Web accessible document depository will be created, but this is not yet available.

5.2 Software

No standard documentation or version control has been implemented, but these will probably use Doxygen and CVS.

6 Development Tools

Tasking's Embeded Development Environment and cross-compilers for C and Assembly Language [5] are currently used at all the CARMA sites. OVRO also has an in-circuit emulator (Ceibo EB-XA/EB-XA-C3, [6]). For testing hardware and algorithms there is an in-house board that plugs into the phyCORE socket on the application board controlled by a PC using Visual Basic or LabVIEW through a DIO24 digital I/O card.

7 References

[1] D. P. Woody, "Proposed CANbus message ID protocol," OVRO, Caltech 26 Mar. 2001.

http://www.ovro.caltech.edu/ovrodocs/CANbus/CANbus_Message_ID_Protocol.pdf

[2] Phytec Elektronik GmbH, "phyCORE-XACx Hardware Manual," June 2000.

[3] S. J. Fredsti, "Software description and theory of operation: Real time operating system for the Philips XAC3 Microprocessor and the Phytec phyCORE XAC3," OVRO, Caltech 25 Jan. 2002.

http://www.ovro.caltech.edu/ovrodocs/CANbus/OVRO_XAC3_RTOS_Theory_of_Operation.pdf

[4] S. J. Fredsti, "Software description and theory of operation for drivers, common utilities and test monitor," OVRO, Caltech 28 Jan. 2002.

http://www.ovro.caltech.edu/ovrodocs/CANbus/OVRO_XAC3_Drivers_&_Common_Software_Theory_of_Operation.pdf

[5] http://www.tasking.com/products/XA/index.html

[6] http://www.ceibo.com/4/ds-xa.shtml