DEVICE NET IMPLEMENTATION UNDER LINUX FOR USE IN CONTROL SYSTEM OF A PARTICLE ACCELERATOR

A. Chepurnov, D. Komissarov, F. Nedeoglo, A. Nikolaev
Institute of Nuclear Physics, Moscow State University. 119899, Moscow, Russia

Abstract

A control system (CS) of an «industrial type» for a new electron linac has been designed. CAN-bus was selected as well as a basic fieldbus for the front-end level of the CS and for operator's console level. To ensure the ability to use components from other developers, the DeviceNet high level protocol for CAN-bus has been selected. To develop the DeviceNet protocol stack Linux has been chosen as the basic operating system. The DeviceNet protocol stack has been developed. The stack contains both Slave's and Master's functionality. The following features are supported today: duplicate MAC ID check, Predefined Master/Slave Connection Set, Unconnected Message Manager for dynamic management of connections. The DeviceNet software package consists of three components. The first component is a configurable, scaleable and portable kernel that contains protocol algorithms, written in pure ANSI C. The second component is a system driver that provides system specific functions to the kernel. The third component is a CAN-bus driver, that provides an interface between the kernel and a particular CAN-bus controller. Currently our DeviceNet system has been tested on Intel-Linux machines equipped with an ISA–CAN interface card. Various utilities have been developed to help management of the DeviceNet network.

1 INTRODUCTION

CAN-bus together with related software and hardware technologies have become more and more popular for development of accelerator control systems of various scales. It is recommended for use in CERN [1]. CAN-bus is used for interconnection between front-end controllers at BESSY II to connect smart I/O system [2]; at DESY to communicate with General Purposes Fieldbus Controllers [3]; at TRIUMF for distributed power control [4] and in some other accelerator centers.

A few years ago we selected CAN-bus as a basic fieldbus. We used it for upgrading our old control system and for development of new control systems for small-scale industrial-type electron linacs [5].

The DeviceNet high level protocol for CAN-bus has been chosen from the three most widely used protocols (CAL/CANopen, DeviceNet and SDS) which are used over CAN-bus in industrial and automotive applications. These protocols are very strongly supported commercially. Specifications are open and available for development of compliant devices and application software. Advantages of the open specification are obvious as strongly as many manufacturers support this specification in their products. We like the Linux OS and prefer to use it in our control system [6].

To our surprise we found no appropriate software components under Linux OS to support CAN-bus and no software components to support DeviceNet. So we decided to develop own software. A DeviceNet compliant protocol stack and different CAN-bus device drivers were developed. The stack can be used to provide Slave capabilities for various types of front-end controllers and Master capabilities for host personal computer (PC) running under Linux OS.

1.1 Object Model

Our implementation of DeviceNet is based on the specification [7]. DeviceNet is defined using object modeling. The Object Model provides a template to implement the Attributes (characteristics of object representation by values), Services (methods or procedures that an object performs) and Behaviors (responses of object to particular events). Graphical representation of an example of the DeviceNet Object Model is presented on the Figure 1.

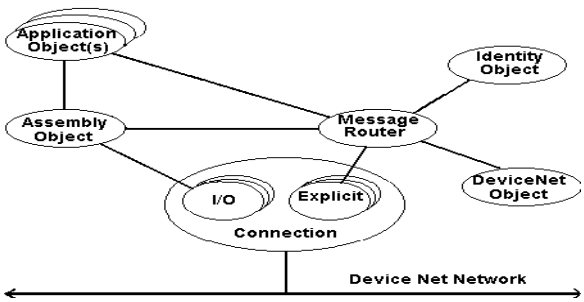

Figure 1: The DeviceNet Object Model.

The *Identity Object* encapsulates information about a Vendor ID, a Device Type, a Product Type, a version, a serial number and product name. This object supports *Get_Attribute_Service.* It means that data about the object are accessible from the network. *Message Router Object* passes Explicit Messages to other Objects. It is used internally and is invisible from the DeviceNet network. The *DeviceNet Object* encapsulates information about a Node Address or MAC ID, a baud rate, a Bus-Off action, an allocation choice, etc. This object supports *Get_Attribute_Service.* The purpose of *Assembly Objects* is to group different attributes from different *Application*

Objects into a single attribute, which can be moved across a single connection. *Connection Objects* represent end points of virtual connections between nodes on the DeviceNet network. There are two types of connections called Explicit Messaging and I/O Messaging. Explicit Messages are used to activate services of the DeviceNet Objects. I/O Messages are used for fast transmission of I/O data across the DeviceNet network -- the procedure that is most popular for the front-end level of the control system. *Application Objects* implement application specific behavior of the device, which allows one to implement control and acquisition algorithms locally.

The model provides an addressing scheme using four numbers - Node Address (MAC ID), the Object Class Identifier, the Instance Number, and the Attribute Number.

Object Model proposed by the DeviceNet specification is convenient for describing and implementing both control and data acquisition algorithms for accelerator control.

1.2 Communication Protocol

The DeviceNet Communication Protocol is based on the idea of connections. To transfer data a device should establish a connection with another device. To establish a connection, each DeviceNet member should include either an Unconnected Message Manager function (UCMM) or an Unconnected Port function. UCMM allows one to create and destroy dynamically Explicit Connections whereas Unconnected Port allows the Master device to allocate a predefined set of connections in the Slave device.

Either UCMM or the Unconnected Port is used to establish an Explicit Connection, which is used then to move data from one node to another, or to establish additional I/O Connections, which allows moving I/O data over the network.

2 DESIGN ISSUES

2.1 General Considerations

All the designed software components are not intended to work in the multithreaded environments.

The DeviceNet protocol stack adheres to the following principles:

- **compatibility** with DeviceNet specification;
- **portability** over various platforms;
- **real-time** capabilities;
- **adaptability** to possible code modification.

The C (ANSI C) language was chosen as a main implementation language for this project for several reasons:

- High level of platform independence of written code, since C compilers exist for a variety of platforms, from PC to single-chip microcontrollers.
- DeviceNet Objects can be modeled easily by means of C structures and functions that manipulate the structures. Since there is no hierarchy in the DeviceNet Object Classes relationship, there is no need to use such Object Oriented languages as C++.

Various versions of the protocol stack software have been tested on the PC under Linux, on micro-controllers from Microchip (PIC16C7x) and on DSP from TI (TMS320C2xx).

Algorithms used in our software have the guaranteed execution time. It is achieved by means of a technique of direct indexing and asynchronous loop exiting.

Direct indexing is used in the parts of the code that operate with data structures contained in arrays. For example, «connection data structures» and pool of message identifiers are both represented as arrays of fixed length. Direct indexing allows us to find any data very quickly within the guaranteed time period. To simplify algorithms and to prevent potential memory leaks the protocol stack software doesn't use heap memory allocation.

Asynchronous loop exiting is used in wait cycles when the program waits for an event to occur. It is useful to implement *Duplication Mac Id Check* algorithm, which must execute every device when it connects to the CAN-bus. When a node wants to connect to the DeviceNet net it sends the special message with broadcast properties and waits for a response. If the response was not received at a fixed time (1 sec), a node may pass to on-line state and perform communication over the net.

2.2 Implementation under Linux

To support CAN-bus-ISA adapters under the Linux kernel a mode driver has been developed [8]. This driver makes the CAN-bus accessible for user level applications by means of writing and reading messages to or from special character device file. The driver can handle up to 4 CAN-bus-ISA adapters simultaneously.

The next implemented component was a Middle Layer Library providing a simple and efficient interface between applications and particular CAN-bus driver. The library makes it possible to implement higher layer software in a cross-platform way. The Middle Layer Library provides the same interface for the application layer for all supported platforms.

The third component was the DeviceNet protocol stack library. A control application can use services from this library to become DeviceNet compliant. A draft version of Master applications based on this library has been tested successfully but the whole Master's functionality is not completed yet. Our implementation of Slave software is completely functional so with the help of this library we can develop simple *Group 2 Only Slaves* and *Slaves with UCMM*. Layered structure of the DeviceNet software system under Linux is shown in Figure 2.

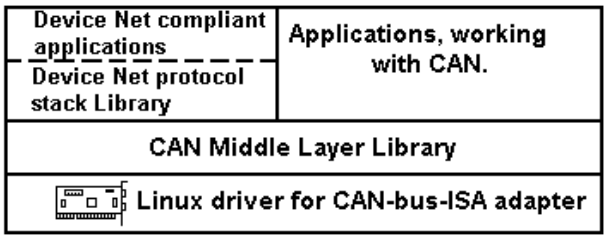

Figure 2: Layered structure of the DeviceNet software.

2.3 DeviceNet protocol stack Library

The library consists of the following parts:

- the library kernel;
- a module with system dependent functions;
- a module with interface to CAN-bus.

The structure of the library is presented in Figure3.

This partitioning scheme ensures portability of our software to platforms with poor resources (micro-controllers) as well as platforms with rich resources such as PCs. The Library kernel contains the protocol stack only and interface to the application program. All dependencies to particular environments are located in the two other parts of the library. So if ones kernel module has been debugged and tested under Linux, it can be used in DeviceNet compliant devices, developed for other platforms. To port the library to a new platform the only thing necessary is to provide system dependent functions (module called *Sysdrv*) and interface to CAN-bus (called *Candrv*).

The approach of separating generic code from highly platform specific code has already been proved. For instance at the beginning of the library development and testing process non-real CAN-bus has been used. The CAN-bus emulator for Linux was used to communicate with nodes over a virtual network. A program which was tested under the CAN-bus emulator can be transformed to a program that works with the real CAN-bus just by relinking its object file with the appropriate *Candrv* module, that implements an interface to the real CAN-bus hardware.

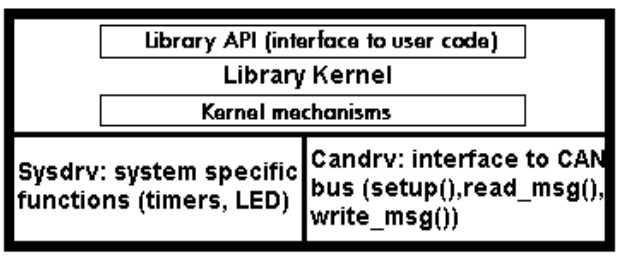

Figure 3: The structure of the DeviceNet library.

3 MAIN RESULTS

Several software projects have been completed or are at an usable stage of development as a result of our effort to create a DeviceNet compatible system. They are: a driver under Linux for CAN-bus-ISA adapter; a CAN-bus emulator for Linux, CAN-bus monitors for DOS and Linux; a low-level configuration utility “dnterm” for DeviceNet nodes manipulation using Explicit Messaging; a knob test program and test I/O programs. The last two projects are based on our DeviceNet library which exists in two variants.

The first variant is for a PC or high-end micro-controller. It supports Explicit, I/O messaging, UCMM, Unconnected Port, fragmentation/reassembly, dynamic management of I/O connections. *The Group 2 Only Client* functionality is supported too. The maximum code size of this version of the library without application code is approximately 25 Kb.

The second one is for simple microcontrollers such as PIC16C7x. It supports only Slave functionality and doesn’t support UCMM. These restrictions simplify library algorithms so its code fits in 4 Kb ROM and its variables require only 100 bytes of RAM.

REFERENCES

[1] G. Baribaud, R.Barillere, A.Bland et al. Reccomendations for the Use of Fieldbuses at CERN in the LHC Era. //Proceedings of the 1997 Int. Conf. on Accelerator and Large Experimental Physics Control Systems, p.285 (Beijing, China, November 3-7, 1997).

[2] J. Bergl, B.Kuner, R. Lange, at al, CAN: a Smart I/O System for Accelerator Controls- Chanses and Perspectives. //Proceedings of the 1997 Int. Conf. on Accelerator and Large Experimental Physics Control Systems, p.290(Beijing, China, November 3-7, 1997).

[3] A. Matioushin, , A Sytin, M. Clausen at al. A Configurable RT OS Powered Fieldbus Controller for Distributed Accelerator Controls. //Proceedings of the 1997 Int. Conf. on Accelerator and Large Experimental Physics Control Systems, p.321 (Beijing, China, November 3-7, 1997).

[4] D. Bishop, D. Dale, H. Hui at al. Distributed Power Supply Control Using CAN-bus //Proceedings of the 1997 Int. Conf. on Accelerator and Large Experimental Physics Control Systems, p.315 (Beijing, China, November 3-7, 1997).

[5] A. Chepurnov, A. Alimov, D. Ermakov, V. Shvedunov Control system for new compact electron linac. // This proceedings.

[6] F. Nedeoglo, D. Komissarov; A.Chepurnov Linux and RT-Linux for accelerator control - pros and cons, application and positive experience. //This proceedings.

[7] DeviceNet Specifications Volume 1, Release 2.0, Volume 2, Release 2.0.

[8] «The Linux Kernel» David A. Rusling, Linux Documentation Project, 1997.