Sample project documents
Sample project documents ................................................................. 1
Project1  CAN SW Design Document .................................................. 1
Project2  Alerting system for evacuation in a Secure CITI - Design Document .......... 10
Project3  Embedded Design – Keypad interfaced, temp and step motor controller ... 14
Project4  WIRELESS SPEAKER --Design and Implementation .......................... 17

Project1  CAN SW Design Document

| Designed by N Krishnamurthy under the supervision of Brad Petrus | Iteration 1.0, February 25, 2007 |
| Bosch Pittsburgh RTC | |

CAN SW Design Document ........................................................................................................... 1
Background: ............................................................................................................................... 2
Design Requirements: .................................................................................................................. 2
  Higher layer Header design: .................................................................................................... 3
  Driver Design Overview: ......................................................................................................... 5
  Interacting Objects .................................................................................................................. 6
Interface Design – Sequence Diagrams ......................................................................................... 6
State Machine of TX/RX threads in daemon .............................................................................. 8
State Machine of TX/RX driver ................................................................................................. 9
Reference: ................................................................................................................................. 9

N. Krishnamurthy

Projects Sample
**Background:**
The Prism middleware (MW) jointly developed by University of Southern California and BOSCH RTC Group, is an architectural software design with abstractions that facilitate distributed software development. The core features provides interfaces for creating components, ports and connectors. The overall composition of a subsystem is specified by the architecture interface which wires up the different constituents.

In addition to providing an event based framework with worker threads responsible for scheduling, dispatching and routing of events between components. Currently the Prism extensions provide OS abstractions of TCP/UDP sockets, and serial communication capability. The objective of this project is to add communication over the CAN bus, and develop test cases as a proof of concept of the functionality of the Prism MW in using the CAN bus.

**Design Requirements:**
1. To use the USB adaptors developed by PEAK-Systems ([http://www.gridconnect.com](http://www.gridconnect.com)) and their driver API’s to develop a multithreaded higher layer.
2. The higher layer (HL) must take care of fragmentation and assembly of the [0-8] bytes of the data link frames provided by the CAN bus. The HL must be able to detect incomplete segments after assembly (in the event the bus failure) and discard them.
3. The HL must be able to multiplex/de-multiplex parallel connections to the CAN bus.
**Higher layer Header design:**

The frame format was thus chosen to address priority, multicasting, muxing/demuxing of received fragments, detection of out of sequence fragments and last fragment in a segment.

The middle frame in figure 1.0 is the total data link (DL) frame of a CAN bus. The 11 bit arbitration field is used to provide explicit priority based arbitration when two or more can controllers access the bus simultaneously. 0 bit is dominant over 1 which is recessive thus the choice of the arbitration field enforces an implicit priority.

The frame on top of the DL frame is our design of the allocation of the 11 bit identifier. Explicit priority of 4 classes is enforced by allocating the first 2 bits. Two groups are currently addressed by the third MC-multicast bit, with unicast addressing given lower priority over group addressing. Note the sender address/port precedes the receiver address/port thus giving transmitters a higher priority.

The HL muxing and demuxing in the use of the can bus is accomplished by using the notion of ports in addition to host addressing. An endpoint source or destination is thus not only identified by the host address but also the port on which the application issued a request or is expecting a reply.

The MF flag bit is used to signal the receiver that more fragments in a segment are on its way. The last fragment has its MF flag unset signaling the end of the segment.

The seven bit sequence numbers are used to detect out of order fragments and to signal to the driver of the receive error if any fragment in a segment is lost.
Figure 1.0 - Header design that addresses:
priority, multicasting, muxing/demuxing, out of sequence fragments and last fragment in a segment

<table>
<thead>
<tr>
<th>Priority</th>
<th>Multicast (MC)</th>
<th>Sender Address</th>
<th>Receiver Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>1 bit</td>
<td>2 bits</td>
<td>2 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

*Explicit 4 priority classes
*Group addressing or multicast feature possible but takes higher precedence to unicast
*mux/demux using <ID/Port> combination
*MF- If set signals more fragments on the way, unset for last fragment
*Seven bit sequence #, to detect out of sequence fragments

Data Link Frame:
*11 bit arbitration, 0-dominant, 1-recessive
*RTR remote transmit request - defaults to 0; if 1 frame is a RR
*DLC indicates # of data bytes
*16bit CRC-error detection
*Ack bit- flanked by a delimiter recessive bit is pulled low by receiver on successful frame reception

Data Link Frame:
<table>
<thead>
<tr>
<th>MF</th>
<th>Seq. #</th>
<th>Data</th>
<th>CRC</th>
<th>ACK</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>7 bits</td>
<td>7 bytes or 56 bits</td>
<td>2 bytes</td>
<td>2 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>
**Driver Design Overview:**

The functionality of the different modules can be abstracted into objects and further into a layering scheme as shown below.

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN Driver D[0-15]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D0</th>
<th>D1</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>D14</th>
<th>D15</th>
</tr>
</thead>
</table>

PCAN Daemon

PCAN-USB API

1. The driver abstraction provides an interface to the application layer to send and receive messages over the CAN bus. Multiple instances of the driver can exist for multiple applications/processes to transmit and receive simultaneously.

2. A singleton driverDaemon abstraction allows for the multiple driver instances to register with it to use the lower level API calls of the pcan_usb.dll.
   a. The daemon implements fragmentation and reassembly of higher layer messages using the API’s provided by the pcan_usb.dll. The fragmentation and reassembly can be implemented as multiple vectors, with a one to one mapping between the registered driver and the vector.
   b. The daemon is also responsible for registering and unregistering the different driver instances. The registration process initializes the vectors to be used with the associated driver.
   c. The reverse mapping of received fragments to buffers is obtained by decoding the header of the received fragment. The CAN controller filter MASK is enabled to receive broadcast – emergency message packets, and packets addressed to self.
   d. D0 is configured by default as the driver to handle broadcast messages, and any host application on power up registers at least this driver.
   e. The daemon inherits the callback feature to notify the driver on successfully assembling an entire segment.
   f. Out of sequence fragments signals a segment loss and the received segment is flushed. The transmitter can be notified using an RTR frame, for now -- to keep the implementation simple we do not support retransmission.
   g. Three threads are envisaged for simultaneous read, write and cleanup operations performed by the daemon. The cleanup thread flushes the receive vector after its individual timeout.
   h. Tx and Rx threads use individual queues to multiplex and demux the use of the underlying CAN bus by a multithreaded application. Note: The
Prism MW which was originally developed in Java has been ported to C++, it has classes defined for threads, mutexes, semaphores, queues and other OS abstractions.

i. The above constructs for threads and mutexes are to be used and restrained use of Standard Template Library (STL), namely queues, lists, vectors would be used to demonstrate the use of CAN in the middleware framework.

j. The lower level API calls are wrapped with mutual exclusion constructs.

See the interface design section for the sequence of function calls between the different objects in the system

**Interacting Objects**

![Interacting Objects Diagram]

The class hierarchy of application and driver for now is kept simple, to a single class to prevent over-objectification. If sufficient properties and behavior distinction of receiver and transmitter is observed upon implementation. A class hierarchy with the common features being exported to an application/driver super class, with specifics of transmission/reception being the extended to respective subclasses can be enforced.

**Interface Design – Sequence Diagrams**

The sequence diagram is made to expose the interfaces of the implementation in terms of function calls. Note: Interfaces have to be clearly identified before actual implementation.
Sequence of function calls between application, driver, daemon and PEAK API, in the envisaged multithreaded CAN support in Prism middleware -- transmit message or write operation.
**State Machine of TX/RX threads in daemon:**

A vector/List of bufferState structure(struct) is used for every transmit/receive buffer. The buffer to driver mapping is obtained (using registration info for Tx or header decoding for Rx fragment), the bufferState struct is passed by reference to the transmit and receive state machines thus pre-empting duplication of state machine code.

```c
typedef struct {
  char * pBuff;
  int offset;  //used by Read/Write thread to update
  time creation;  //used by cleanup thread to time out and flush segment
  int state
} bufferState;
```

![Driver Daemon - Receive/Transmit Thread Event Handling](image-url)
**State Machine of TX/RX driver:**

![State Machine Diagram]

**Reference:**

[1] BOSCH CAN Standard 2.a and 2.b  
[2] TCP/IP Illustrated Volume 1 W. Richard Stevens  
Background:
The envisaged secure citi(S-CITI) users are all equipped with Zaurus SL 5500 pda’s that are wireless enabled and run Linux 2.4.18. The objective of this project is to implement an alerting mechanism that runs on top of ($\alpha$,t) protocol framework that supports different inter and intra cluster routing. [1] have developed a user space Distributed Dynamic Algorithm (DDCA) which sets up the routing framework i.e routing tables are updated dynamically by discovering and maintaining a neighbor list through periodic hello messages.

The alerting application is to use the abovementioned framework to communicate the evacuation plans to different users based on the number of users in a room(s) and the exits available to them. The location problem of the pda users in the building is solved by giving the users a list of rooms to pick from and an option to switch rooms as they move in the building. The server thus keeps track of the different users in the building and not only delivers individual evacuation plan, but also provides an additional chat feature ☺ for the users within a room.

System Requirement:
The system requirements are broken down to requirements at the client and the server.

Client side requirement:
1. Clients must have a user interface to execute commands.
a. to view available rooms
b. to register to one of the rooms to receive alert messages (could also be chat messages from users in the room – future?)
c. to switch between rooms as they move in the building
d. to quit the application when they exit the Secure CITI

Server side requirement:
1. Server must have a configuration file of the available rooms that a client can register to.
   a. Must support multiple clients simultaneously, and maintain location information of individual clients
   b. Facilitate register-response mechanism for the location registration process
   c. Must be amenable to quickly implement the chat feature for users within a room once the high priority alert mechanism is in place.

System Architecture:

\[
\begin{array}{c}
\text{Distributed clientServer info process (TCP)} \\
\text{Cfg file - memberInfo} \\
\text{User Interface and (Chat session)} \\
\text{Named pipe IPC}
\end{array}
\quad
\begin{array}{c}
\text{TCP Server, tear down connection after every msg.} \\
\text{Cfg file - rooms}
\end{array}
\quad
\begin{array}{c}
\text{UDP Server for chat}
\end{array}
\]

Rationale behind the architecture:
The system requirement of the S-CITI system reflects a striking similarity to a widely prevalent application on the internet – the chat system. The architecture for such a system with suitable modifications would be an ideal design for our S-CITI alerting system [2].

The clients contact the server with their information <host><port> etc. (see section on tcp messages) and the room they would like to register, the server looks up the room info; if capacity is not reached it sends a response OK for the registration if not it rejects it.
response carries a randomly generated ID for each client that enhances security, which the client uses for the ensuing communication with the server.

The User Interface (UI) is a process forked by the client’s Server info process (cSip), since the Linux pda’s don’t have xterm on them we cannot fork and exec xterm –e userIface. For now we have to run two processes at the client one for the - UI where the alert and chat messages would be received. Additionally the UI is where the user issues commands to register/switch rooms. The second process in the client - cSip is the distributed server information store and TCP communication interface.

Decomposing the system functionality into UI and cSip ensures modularization and ease of code maintenance for future feature additions. The communication between cSip and the server uses TCP sockets, with connection establishment and tear down after the message is communicated (this is similar to http), this ensures that the connection states don’t clog the server process.

cSip distributes the server role – plays an important part in the p2p chat system, it caches the peer clients <host> <port> information that is obtained from the server which it uses directly, instead of querying the server to send UDP chat messages.

**Control messages between UI via cSip and server**

The end of any message is indicated using the new line character (‘\n’), following are some of the messages for the envisaged system.

- **JOIN REQUEST** `<memberofname> <hostname> <udpPort> <tcpPort> \n`<memberofname> is the nickname the member wishes to use.<hostname> is the name of the machine on which the client is running.<udpPort> is the UDP port of the client where chat messages may be sent.<tcpPort> is the TCP port of the information server where information requests must be sent.

The client registers to the server using this message and must send this message first, before sending any other messages.

- **SWITCH ROOM REQUEST** `<memberID> <roomName> \n`<memberID> is the ID assigned to the member. (Refer to JOIN RESPONSE below)<roomName> is the name of the chat room the member wishes to switch to.

- **JOIN RESPONSE OK** `<memberID> \n` (or)
- **JOIN RESPONSE REFUSE** `<memberID>`

<memberID> is a randomly assigned ID that is used to identify this client in subsequent messages.

The server can deny the client permission to enter the system/room either because the server limits the number of users in a room and prevents duplicate membernames within a room.

- **SWITCH ROOM RESPONSE OK** `<memberID>` (or)
- **SWITCH RESPONSE REFUSE** `<memberID>`

The server can deny the client permission to switch to a different room, for similar reasons given above.

- **MEMBER LIST REQUEST** `<memberID>`

This message asks for a list of all members logged on to the chat session.

- **MEMBER LIST RESPONSE** `<space delimited list of member nicknames>`
• ROOM LIST REQUEST
This message asks for a list of names of all chat rooms.
• ROOM LIST RESPONSE <space delimited list of rooms>

• QUIT MSG <memberID>
<memberID> is the ID assigned to the member.
This message is sent when the client terminates the chat session.

These messages are not exhaustive, and do not contain all the chat related messages some of which are listed below:

• MEMBER INFO REQUEST <remoteMemberName>
<remoteMemberName> is the nickname of the member whose information the client wants.
• MEMBER INFO RESPONSE INFO FOLLOWS <hostName> <TCPPort> (or)
• MEMBER INFO RESPONSE UNKNOWN MEMBER
<hostName> is the name of the machine on which the remote client is executing.
<TCPPort> is the TCP port of the information server of the remote client.
If no member exists with the requested the server replies with a UNKNOWN MEMBER response. Otherwise, the server returns the <host> and <port> of the remote client.

Implementation Concerns:
1. Server multiplexes many clients using the FD_SET feature of select system call.
2. Named pipes or FIFO is used for communication between UI and cSip processes, have to look into the details of using mknod – block/char special files for implementing FIFO before proceeding.

References:
[1] On the implementation and performance of the (α,t) protocol on linux A Gopalan, S Dwivedi and T Znati
[2] Man pages and WWW for the innumerable examples on linux socket programming, e.g
http://beej.us/guide/bgnet/
http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html
Project 3: Embedded Design – Keypad interfaced, temp and stepper motor controller

Project 3a: Design of a Keypad User Interfaced temp/light controller:

Sequence and Collaboration diagram for: LOGON,

Implemented a state-machine, in the foreground for prompting the user with a display menu, while ISR decoded the keyed digits.

Projects Sample
Project 3b: Design of a multi-process temperature simulator:

Project 3a was extended to implement multitasking with priority:
The temperature simulator module was written to increment a seed temp setting value by two degrees, the temp setting values were stored in a circular array, with the array index incremented under a modulo operation (i.e. array index reset to zero upon reaching the length of the array). A timer was implemented to trigger multiple events, a 2 sec simulator temp increment event and a 10 sec system time display events in addition to the asynchronous user event to display the avg temperature of the last 4 temperature of the simulator.

Multitasking and task synchronization, signaling using semaphores, circular buffer implementation enabled us to integrate individual modules with minimal changes to support intertask communication/signaling.

Priorities of various tasks are:
- User input ISR – asynchronous highest priority
- Ten second system time display – priority 1
- Temperature simulator – priority 2
- Average temp calculation and display of latest four temp settings – priority 3
- User interface menu – priority 4
- Bubble sort task – lowest priority.

Results:

<table>
<thead>
<tr>
<th>Time Task_system_time:</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator Task:</td>
<td>253</td>
</tr>
<tr>
<td>incrCnt:1 tempVal:125</td>
<td>tickCnt:253</td>
</tr>
<tr>
<td>init random array</td>
<td></td>
</tr>
<tr>
<td>sorted first 10 nos:</td>
<td>0_1_1_3_3_3_6_7_8_8</td>
</tr>
<tr>
<td>Time Task_system_time:</td>
<td>2000</td>
</tr>
<tr>
<td>Simulator Task:</td>
<td>2253</td>
</tr>
<tr>
<td>incrCnt:1 tempVal:127</td>
<td>tickCnt:2253</td>
</tr>
<tr>
<td>the digit is: 7</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:127</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:125</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:0</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:0</td>
<td></td>
</tr>
<tr>
<td>Tick:2304 Avg Temp Task Avg_temp is: 63</td>
<td></td>
</tr>
</tbody>
</table>

Press 7 for view, Press 8 for Set
Time Task_system_time: 6000
Simulator Task: 6253
incrCnt:3 tempVal:131 tickCnt:6253
the digit is: 8
Enter 3 digit Temp Setting
the digit is: 4
the digit is: 5
the digit is: 5
Press 7 for view, Press 8 for Set
the digit is: 9
Time Task_system_time: 8000
Simulator Task: 8253
incrCnt:5 tempVal:457 tickCnt:8253
the digit is: 7
the array avgTemp:457
the array avgTemp:455
the array avgTemp:131
the array avgTemp:129
Tick:9310 Avg Temp Task Avg_temp is: 293

<table>
<thead>
<tr>
<th>Time Task_system_time:</th>
<th>8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator Task:</td>
<td>8253</td>
</tr>
<tr>
<td>incrCnt:2 tempVal:129</td>
<td>tickCnt:4253</td>
</tr>
<tr>
<td>the digit is: 7</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:129</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:127</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:125</td>
<td></td>
</tr>
<tr>
<td>the array avgTemp:0</td>
<td></td>
</tr>
<tr>
<td>Tick:4453 Avg Temp Task Avg_temp is: 95</td>
<td></td>
</tr>
</tbody>
</table>
Project 3c: Design of a Keypad User Interfaced Stepper Motor controller:

nios-run: Terminal mode (Control-C exits)
-----------------------------------------
Enter 1 to chg speed_dir
the digit is: 1
enter 8-clockwise 9-anticlock
the digit is: 1
the digit is: 5
the digit is: 6
the digit is: 8
new speed setting:1568
curspeed:0 newspeed:1568
dacval:14 dacincr:0 dacease 128

Note: The bit value of currents pd is the digital input to the DAC:
- 255 bits corresponds the maximum speed in the clockwise direction, while
-128 bits is the zero speed and
-0 bits corresponds to a maximum speed in the anticlockwise direction
Project4 WIRELESS SPEAKER --Design and Implementation

Design Sallen-Key Band Pass filter

Design of IR AM transmitter-receiver

IR AM Transmitter

Audio signal from CD player

IR AM Detector

Signal to Filter, Preamp & Power Amp

Photo transistor

Design of Power supply and Current booster

N. Krishnamurthy