CAN bus Data Stream Wrapper

YANG YANG
Abstract

CAN bus Data Stream Wrapper

YANG YANG

A data stream management system (DSMS) is similar to a database system with the difference that a DSMS can search data directly in on-line streams as well as querying stored data, while a DBMS can search only stored data. Stream queries are called continuous queries because they run all the time until they are terminated. SCSQ is an extensible DSMS allowing different kinds of data sources to be integrated and queried. A SCSQ interface to a data stream system is called a datastream wrapper. A datastream wrapper allows continuous queries to be specified over an external data stream producing system. The Controller Area Network bus (CAN bus), is a standard for interfacing data streams from different kinds of equipment and engines, such as wheel loaders and other vehicles. The objective of the project is to develop an interface, called a CAN bus datastream wrapper, to enable SCSQ to access to streams of sensor readings from industrial equipment through CAN bus standard interfaces. It enables the SCSQ user to specify continuous queries over equipment data streams.
Content

1 Introduction ............................................................................................................. 2
2 Background .............................................................................................................. 4
   2.1 Database manage systems (DBMSs) ................................................................. 4
   2.2 Data Stream Management Systems (DSMSs) ..................................................... 5
   2.3 The CAN bus .................................................................................................... 6
   2.4 SCSQ and Amos II ........................................................................................... 10
   2.5 The Kvaser CAN Bus Software .......................................................................... 11
      2.5.1 The Virtual Driver ...................................................................................... 11
      2.5.2 CanKing and Traffic Generator ..................................................................... 11
      2.5.3 The Kvaser SDK .......................................................................................... 13
3 The SCSQ CAN bus data stream wrapper ............................................................... 15
   3.1 The structure of wrapper .................................................................................... 15
   3.2 Implementation ................................................................................................ 17
      3.2.1 canreadrecords() ....................................................................................... 17
      3.2.2 canreadbag() .............................................................................................. 18
      3.2.3 canRead ....................................................................................................... 20
      3.2.4 canOpenChannel ......................................................................................... 21
      3.2.5 canClose ..................................................................................................... 21
      3.2.6 Error management for SCQS ....................................................................... 21
4 Running the CAN Wrapper .................................................................................... 22
   4.1 Running steps .................................................................................................... 22
   4.2 Process the CAN bus data by AMOSQL ........................................................... 23
5 Future work ............................................................................................................ 26
References ................................................................................................................. 27
1 Introduction

In many contexts, there is an increasing need for collecting different kinds of measurement data from various machines for performance monitoring, diagnostics, and fault detection. For example, modern industrial equipment such as wheel loaders generate large volumes of digital data logs while in operation. These logged data is a valuable asset for the equipment manufacturers, for purposes of diagnostics, testing, and verification. By continuously reading and analyzing various parameters of sensor data from machines in use over time, a system can identify when the equipment is not working as expected, i.e. non-normal behavior.

As database management systems (DBMSs) [Elmasri&Navathe], data stream management systems (DSMSs) [1] process data using queries. In contrast to database management system, which executes queries over stale data stored in a database, a DSMS executes continuous queries (CQs) over continuously flowing data streams, such as streams from wheel-loaders, satellites, or stock feeds. A CQ is a query that once it is issued over a datastream $D$ runs continuously over the data produced by $D$ until the CQ is terminated [2]. This is different from regular database queries, which return the complete results immediately. The majority of DSMSs are data-driven so that whenever new data arrives, a continuous query produces new results tuples.

The Controller Area Network (CAN) bus [3] is a vehicle bus standard designed to allow micro-controllers and devices to communicate with each other within a vehicle without a host computer. CAN bus is widely used in cars, wheel loaders, cranes, and other many fields.

Equipment manufacturers would like to monitor the large number of CAN bus devices of different kinds on installed equipment in order to, e.g., identify faulty equipment. Abnormal CAN bus readings should be reported instantly and logs from faulty CAN bus data logged in files. In this project it is investigated how CAN bus data can be analyzed using CQs to a DSMS. It should be possible to process CAN bus data during long time periods as continuous queries over data in CAN bus data streams.

SCSQ [4] is an extensible DSMS allowing different kinds of data sources and streams to be integrated and queried. To enable analyzing data streams produced by external systems, e.g. through the CAN bus protocol from a vehicle, the user can define datastream wrappers to interface external data streams. A datastream wrapper can be implemented in some regular programming language like C, C++, Java, or Python.
The purpose of this project is to enable continuous monitoring in SCSQ of data produced by the standard CAN bus interface to machine data. The CAN bus interface runs inside equipment such as wheel loaders. It emits streams of sensor readings from the equipment. In the project an interface, called the CAN bus datastream wrapper, between SCSQ and the standard CAN bus is developed. For practical reasons, it is desirable to test out and experiment with the CAN bus datastream wrapper based on simulated CAN bus data. Also it is desirable to reuse existing software to interface CAN bus streams when implementing the wrapper. Therefore the first step was to investigate what tools are available for interfacing CAN bus data and for generating simulated CAN bus streams off-line.

Kvaser [5] supplies advanced CAN solutions to engineers in designing and deploying equipment interfaces. Kvaser CAN drivers [6] supports many kinds of CAN hardware. It also supports the CAN virtual device, which is a device driver, analogous to a virtual network adapter or a virtual CD drive, which simulates one or several real CAN bus devices. It can be installed in Windows and Linux. Kvaser CAN king is a free of charge general purpose CAN device monitor that controls the Kvaser virtual CAN bus and generates simulated continuous CAN bus datastreams. Kvaser provides CANlib SDK [6], which is an API for accessing CAN bus data from application programs in C or C++.
2 Background

Before we start the project of SCSQ CAN bus data stream wrapper, it is necessary to understand some basic conceptions: databases and DBMS, Data Stream and DSMS, know the concept of CAN bus and understand the basic format of CAN data. It is also very important to learn how to use the Kvaser CANbus tools suits, for example Kvaser CANking, Kvaser CAN SDK, Kvaser virtual devices and so on.

2.1 Database manage systems (DBMSs)

A general-purpose DBMS is a software system designed to allow the definition, creation, querying, update, and administration of databases. A DBMS is responsible for maintaining the integrity and security of stored data, and for recovering information if the system fails. Comparing to a file system a DBMS provides several advantages, like low redundancy, high data consistency, high integrity, data security, transaction support, etc. [7].

SQL (Structured Query Language) is a standard computer language for relational database management and data manipulation. SQL is used to query, insert, update and modify data. The SQL language is subdivided into several language elements, including: clauses, expressions, predicates, queries and statements [14].

An example of an SQL query is:

```
SELECT *
FROM Book
WHERE price > 100.00
ORDER BY title;
```

A DBMS is designed for static data in conventional databases and SQL is the most common database query language. DBMSs cannot process data streams like CAN bus data frames.
2.2 Data Stream Management Systems (DSMSs)

A Data Stream Management System (DSMS) is a computer program to query continuous data streams. Traditional DBMSs run one-time queries over finite stored data in conventional databases. However, many modern applications such as network monitoring, financial analysis, manufacturing, and sensor networks require long-running, or continuous queries (CQs) over continuous unbounded streams of data [9], which are provided by DSMSs. A continuous query is a special query that is issued once over a data stream, and then logically runs continuously until it is terminated. In contrast to a DBMS, a DSMS executes a continuous query which is not only performed once but permanently installed for long time periods. Therefore, the query is continuously executed until explicitly uninstalled. Continuous produces new results while new data arrive at the system simultaneously.

One of the most important features of a DSMS is the possibility to handle potentially infinite and rapidly changing data streams by offering a flexible query processing ability. DSMS can process queries over possibly infinite streams with limited CPU and memory resources.

SCSQ [4] is a DSMS that can process both regular queries over data and meta-data stored in its internal main-memory database and CQs over data in streams, as illustrated by Figure 2.3.

![Fig.2.3 The SCSQ system. Ovals indicate that the local database is stored in the main memory of SCSQ.](image-url)
The following table provides various basic principles of DSMS and compares them to traditional DBMS.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DSMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent data (relations)</td>
<td>Volatile data streams</td>
</tr>
<tr>
<td>Random access</td>
<td>Sequential access</td>
</tr>
<tr>
<td>One-time queries</td>
<td>Continuous queries</td>
</tr>
<tr>
<td>Large secondary storage</td>
<td>Main memory database in SCSQ</td>
</tr>
<tr>
<td>Relatively low update rate</td>
<td>Potentially extremely high update rate</td>
</tr>
<tr>
<td>Little or no time requirements</td>
<td>Real-time requirements</td>
</tr>
</tbody>
</table>

*Table 2.3 comparison of DBMS and DSMS*

### 2.3 The CAN bus

The Controller Area Network (CAN) is a serial communications protocol which can efficiently support distributed real-time control systems with extremely high level of security. Its domain of application ranges from high speed networks to low cost multiplex wiring. [10]

*Fig.2.1 CAN data format of base frame [10]*
<table>
<thead>
<tr>
<th>Field name</th>
<th>length (bits)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start-of-frame</td>
<td>1</td>
<td>Indicates the start of a frame transmission</td>
</tr>
<tr>
<td>Identifier (green)</td>
<td>11 or 29</td>
<td>A (unique) identifier for the data</td>
</tr>
<tr>
<td>Remote transmission request (RTR)</td>
<td>1</td>
<td>Dominant (0) (see Remote Frame below)</td>
</tr>
<tr>
<td>Identifier extension bit (IDE)</td>
<td>1</td>
<td>Declaring if 11 bit message ID or 29 bit message ID is used. Dominant (0) indicates an 11 bit message ID while Recessive (1) indicate a 29 bit message ID.</td>
</tr>
<tr>
<td>Reserved bit (r0)</td>
<td>1</td>
<td>Reserved bit</td>
</tr>
<tr>
<td>Data length code (DLC) (yellow)</td>
<td>4</td>
<td>Number of bytes of data in message (0–8 bytes)</td>
</tr>
<tr>
<td>Data field (red)</td>
<td>0–64 (0-8 bytes)</td>
<td>Data to be transmitted (length in bytes dictated by DLC field)</td>
</tr>
<tr>
<td>CRC</td>
<td>15</td>
<td>Cyclic redundancy check</td>
</tr>
<tr>
<td>CRC delimiter</td>
<td>1</td>
<td>Must be recessive (1), the checksum for the CAN message</td>
</tr>
<tr>
<td>ACK slot</td>
<td>1</td>
<td>Transmitter sends recessive (1) status and any receiver can assert a dominant (0) status</td>
</tr>
<tr>
<td>ACK delimiter</td>
<td>1</td>
<td>Must be recessive (1)</td>
</tr>
<tr>
<td>End-of-frame (EOF)</td>
<td>7</td>
<td>Must be recessive (1)</td>
</tr>
</tbody>
</table>

*Table 2.1 Data field description of base CAN bus frame [10]*

A CAN network can be configured to work with two different message (or "frame") formats: the standard frame format and the extended frame format. All frames begin with a start-of-frame (SOF) bit that denotes the start of the frame transmission.
The identifier defines the content of the message, where different companies have different meanings like temperature, warnings, etc. The identifier can also represent the priority of the message. As illustrated in Figure 2.1 and table 2.1, the "CAN base frame" has 11 bits for the identifier that represents the type of the message. The "CAN extended frame" has a 29 bits identifier, made up of the 11-bit identifier ("base identifier") and an 18-bit extension ("identifier extension"). CAN controllers that support extended frame format messages are also able to send and receive messages in CAN base frame format.

CAN has four frame types:
1. A data frame is the most common message type. It contains message data which carries information for transmission which is actual data from source node in the equipment to a destination node. In our case, the source node is a virtual can bus interface and the destination node is a DSMS.
2. A remote frame requests the transmission of a specific identifier in the inverse direction from a destination node to a source node to request data to control the remote node.
3. An error frame is transmitted by some node to detect an error. Error frames can be transmitted both in source-destination and destination-source direction.
4. An overload frame injects a delay between data frames.

The CAN standard requires that the implementation must accept the base frame format and optionally may accept the extended frame format.

Kvaser supplies different kinds of advanced CAN related products and solutions to engineers. These systems can be used in areas as wide ranging as buses and trucks, electric or petrol-driven cars and construction equipment, avionics, building automation, domestic appliances, industrial automation, telecoms, marine, railway, medical, textiles and more. [5]

Kvaser provides tools used in the project, i.e. a virtual CAN bus interface and a CANking data generator. The virtual CAN bus interface provides a virtual CAN device in a computer. The CANking data generator sends random data to this virtual CAN device as CAN data frames.

SCSQ is suitable for processing continuously produced CAN bus data frames from all devices (virtual or real). Figure 2.3 illustrates how random data streams of CAN bus frames are generated by Kvaser’s CAN Bus Loader.
This project implements a datastream wrapper to query CAN bus data frames produced by CANKing and accessed through virtual CAN bus interface from CQs processed by SCSQ. In our implementation, the source node is a Kvaser virtual can bus interface and the destination node is SCSQ.
2.4 SCSQ and Amos II

Amos II [4] is an extensible NoSQL database system allowing different kinds of data sources to be integrated and queried. The system can store data in its main-memory database. Furthermore, wrappers can be defined for different kinds of data sources and external storage managers to make them query-able. Amos II includes primitives for data mining through functions for data analysis, aggregation, visualization, and handling of ordered collections through the data type Vector. AMOS II can be used a single user database, a multi-user server, or as a collection of interacting AMOS II multi-database peers.

The query language of Amos II is called AmosQL. Similar to SQL, AmosQL is a combination of a DDL (Data Definition Language) and a DML (Data Manipulation Language). The query part of the language is similar to SQL. The following AmosQL query answers the question ‘What country arranged the World Cup 1982?:

```
select x
  from Charstring x, Country c, Tournament t
where x=name(c)
  and c=host(t)
  and year(t)=1982;
```

In AmosQL query variables are bound to the instances of objects of any kind (domain calculus), in contrast to SQL where variables in queries always have to be bound to rows in tables (tuple calculus). Queries in terms of function compositions over sets of objects make AmosQL queries versatile.

SCSQ [16] generalizes Amos II for scalable processing of continuous queries over data streams. The query language in SCSQ is called SCQL. It extends AmosQL with CQs over data streams. Thus in SCSQL variables can range over data streams in addition to data stored in the local database as in AmosQL.

Functions in SCSQ can represent tables in its local database or views over these tables. Foreign functions are implemented in some regular programming language such as C, Java, or Python. A foreign function iteratively computes its result tuples given a set of actual arguments. Foreign functions enable the system to access external data sources in queries. In SCSQ functions can be defined that operate over data streams rather than only data stored in the local database as in Amos II.

The data stream wrappers of SCSQ are defined and implemented as foreign functions over data streams that iteratively emit data tuples from a data stream. Since the result tuples are computed iteratively with foreign functions there is no limitation on the length of the stream. In the project, the foreign function interface
of SCBX is used to build a data stream wrapper to communicate with CAN bus through Kvaser’s CANlib software.

### 2.5 The Kvaser CAN Bus Software

Kvaser CAN bus software is a suit of useful software and SDK and drives. This project will use CANbus virtual driver and CanKing traffic generator and Kvaser SDK.

#### 2.5.1 The Virtual Driver

Kvaser’s CAN driver [6] supports different kinds of CAN devices, e.g., Kvaser LAPcan, Kvaser PCIcan, Kvaser BlackBird, Kvaser Ethercan Light HS, Kvaser Eagle, Kvaser Memorator, Kvaser USBcan II, Kvaser Mini PCI Express HS and so on. Particularly in this project, the Kvaser Virtual CAN Bus is used. The Virtual CAN Bus is very similar to real CAN bus device. With these virtual channels, programmers can begin testing their software conveniently before they have a piece of real CAN hardware. If the drivers of CAN interfaces installed correctly, the OS (Windows in the project) can receive the data in CAN bus format.

![virtual CAN driver](image.png)

*Fig 2.5.1 virtual CAN driver*

#### 2.5.2 CanKing and Traffic Generator

CanKing is a free software produced by Kvaser. It is a general purpose CAN Bus monitor and also a CAN bus random data generator.
The project uses the **CanKing Traffic Generator**, which generates simulated CAN bus data streams. In this project the traffic generator is used for testing the functionality of the developed SCSQ CAN bus datastream wrapper. The CAN bus data generated by the traffic generator can be of different kinds. The CanKing Traffic Generator can produce CAN bus streams of different rates, with different frame ids, and different message data.
The user can use the CAN Bus Loader control panel (Figure 2.5.1) to control the simulated data frames, where:

*Rate* is the frequency of generated CAN bus data. It can be random or constant.

*Burst size* is the size of the generated CAN bus data frame.

*Count* is the number of CAN bus messages. It can be constant or infinite (continuous)

*ID* is the frame identifier.

*Data* is that message data field of generated CAN bus frames. It can be a sequence number or a random number. The length of the data field can also be varied. It can be fixed or random length up to maximal data length (8) of the CAN bus message data field.

Using the CanKing Traffic Generator, different formats of CAN data can be generated that simulate different kinds of CAN data and CAN interfaces. For example, CAN data messages can be simulated with constant 100ms frequency, random frequency from 50-200 ms, constant or random burst sizes, constant or random CAN identifiers, and random data. CANKing Traffic Generator is a good tool for simulation of CAN data used in the project.

### 2.5.3 The Kvaser SDK

CANlib SDK is the software development kit for the Kvaser CAN interface which includes Link libraries, head files, documents and components. CANlib SDK provides the application programmer with a quick and easy access to the CAN interface and CAN network [12].

This project uses the CANlib SDK for Windows which supports Windows XP, Windows 7, and Windows 8. This CANlib SDK supports the programming languages in C/C++, Delphi, .NET 4.0, and C#. The CANlib SDK supports the features of library and channel initialization, go on and off the CAN bus, reading and sending messages, asynchronous notification, messages filters, and so on. This enables the programmer to control the CAN bus channel with different kinds of functions.

<table>
<thead>
<tr>
<th>Core API</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>canInitializeLibrary()</td>
<td>Initialize the library</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>canOpenChannel()</td>
<td>Return a handle to a specific CAN circuit which is similar to file descriptor</td>
</tr>
<tr>
<td>canSetBusParams()</td>
<td>Sets the bus timing parameters for the specified CAN controller</td>
</tr>
<tr>
<td>canBusOn()</td>
<td>Takes the specified channel on-bus</td>
</tr>
<tr>
<td>canWrite()</td>
<td>Sends a CAN message</td>
</tr>
<tr>
<td>canRead()</td>
<td>Reads a CAN message</td>
</tr>
<tr>
<td>canReadWait()</td>
<td>Reads a message from the receive buffer</td>
</tr>
<tr>
<td>canWriteSync()</td>
<td>Waits until all CAN messages for the specified handle are sent, or the timeout period expires</td>
</tr>
<tr>
<td>canBusOff()</td>
<td>Takes the specified channel off-bus</td>
</tr>
<tr>
<td>canClose()</td>
<td>Closes the channel associated with the handle</td>
</tr>
</tbody>
</table>

Table 2.5.3 API of CANlib SDK

By using Kvaser CANlib SDK, the application can communicate with CAN devices. The application can read and write different kinds of CAN data information from or to these CAN interfaces which includes CAN tunnel information and raw CAN data with different kinds of CAN devices. The application can also send control signals to CAN interface, like ON OFF OPEN CLOSE and so on.

In this project, the **SCSQ CAN bus datastream wrapper (SCBW)** is implemented. It implements a SCSQ datastream wrapper that communicates with Kvaser’s virtual CAN bus interface.
3 The SCSQ CAN bus data stream wrapper

The SCSQ CAN bus data stream wrapper is the main work of the project. It is the middleware between SCSQ and CAN devices. It can execute continues queries and read and write data from CAN devices to SCSQ kernel. The SCSQ CAN bus data stream wrapper use CANlib SDK from Kvaser AB. It is implemented in C.

3.1 The structure of wrapper

SCBW enables continuous SCSQL queries to CAN bus data streams, as illustrated by Figure 3.1.

![Figure 3.1.1 CAN bus datastream wrapper architecture](image-url)

We can divide the main architecture of SCBW into three parts:

The **CAN device** can be a real CAN device like PCIEcan, PCIcan, Blackbird, Eagle USBcan [5] and so on. It can also be a virtual CAN device like the CanKing data generator.

The **CAN wrapper** is written as a foreign SCSQ function in C. It uses Kvaser’s **CANlib SDK** to open a CAN bus channel and read the CAN bus data to emit the data iteratively to the **SCSQ kernel** as a data stream of tuples.
The SC SQ system processes continuous SCSQL queries filtering and transforming the received CAN bus data.

The following are two examples of continuous SCSQL queries using the CAN bus data stream wrapper.

```
"ts":1316,"frame":93,"value":6311,"flags":2
"ts":1416,"frame":50,"value":6312,"flags":2
"ts":1516,"frame":17,"value":6313,"flags":2
"ts":1616,"frame":91,"value":6314,"flags":2
"ts":1716,"frame":63,"value":6315,"flags":2
"ts":1816,"frame":42,"value":6316,"flags":2
"ts":1916,"frame":70,"value":6317,"flags":2
"ts":2116,"frame":55,"value":6319,"flags":2
"ts":2216,"frame":68,"value":6320,"flags":2
"ts":2316,"frame":26,"value":6321,"flags":2
"ts":2416,"frame":52,"value":6322,"flags":2
"ts":2516,"frame":50,"value":6323,"flags":2
"ts":2616,"frame":9,"value":6324,"flags":2
"ts":2716,"frame":14,"value":6325,"flags":2
"ts":2816,"frame":7,"value":6326,"flags":2
"ts":2916,"frame":59,"value":6327,"flags":2
"ts":3016,"frame":32,"value":6328,"flags":2
```

Figure 3.1.2 Result from the CQ:
```
sel ect r from Record r where r in canreadrecords(0);
```

The CQ illustrated by Figure 3.2.1 retrieves the raw data stream of records read from the CAN bus steam in channel 0. The screen shot shows a small section of the console output from executing the CQ. The records are represented in SC SQ as JSON objects.
Figure 3.1.3 Result from CQ:

```sql
select r from Record r where r in canreadrecords(0) and r["frame"] > 80;
```

The CQ illustrated by Figure 3.2.2 selects the records read from the CAN bus stream in channel 0 where the attribute “frame” has a value larger than 80.

### 3.2 Implementation

The most important function of the CAN bus datastream wrapper is `canreadbag()`. In order to start `canreadbag` and monitor the data stream of continuous queries, we run `canreadrecords` in the SCSQ which can call `canreadbag()` and gets JSON objects.

Inside the `canreadbag()` function, we have 3 major steps: 1. Initialize and Setup. 2. Read the CANbus data stream and emit to SCQS. 3. Turn off and Close.

#### 3.2.1 canreadrecords()

SCBW is implemented by the function `canreadrecords()` that continuously produces a stream of JSON objects representing CAN bus frames read through Kvaser’s virtual CAN bus interface to some equipment. It is a derived function with the definition:

```sql
create function canreadrecords(Number channel) 
    -> Stream of Record
```
as streamof(select {"ts": ts,
    "frame": fr,
    "value": v,
    "flags": fl}
    from Number ts, Number fr, Number v,
    Number fl
    where (ts, fr, v, fl) in canreadbag(channel));

3.2.2 canreadbag()

The foreign function canreadbag() reads the CAN bus data frames, converts them to
SCSQ tuples, and iteratively emits the tuples to SCSQ. Its implementation in C as
explained below. The emitted tuples have four elements, a time stamp ts, a CAN bus
frame identifier fr, a frame value v, and the CAN bus flags fl. The tuples are
converted to JSON records using the syntax {attribute:value ....}. The JSON
records are returned as a data stream through the stream generator streamof().

The foreign function canreadbag() is implemented in C.
It has the signature:

create function canreadbag(Number channel)
    -> Bag of (Number ts, Number fr,
    Number val, Number flg)
    as foreign 'canreadbag-+++';

canreadbag-+++ means input,output,output,output

In general, the implementation in C code of canreadbag() can be divided into 3 parts.

Initialize and Setup:
Wrapper initialize the CANbus library. Then the wrapper should get the number(id)
of the CANbus channel which get from SCQS input. Then set the parameters of the
CANbus channel and turn it on.

Read the CANbus data stream and emit to SCQS:
Read the CANbus stream message to the buffer, if the communication is fine, emit
the message to the SCQS continuously. If the status of the CANbus channel is not OK,
report en error.
The code of canreadbag() in C is shown below:

```c
void canreadbag(a_callcontext cxt, a_tuple tpl)
{

/* Initialize and Setup*/
  int chan = (int)a_getintelem(tpl, 0, FALSE);
  int hnd=InitCtrl(chan); // Initialize CAN Bus channel
  canStatus stat; // CAN bus status
  unsigned long timeOffset = canReadTimer(hnd); // Get Unix time stamp of CAN Bus stream start
  long id; // a frame identifier
  unsigned int dlc; // length of frame vale
  unsigned int flags; // xxx flags
  unsigned char msg[8];
  LONGINT *datap; //msg as 64 bit integer
  DWORD time; // time from start in ms

/*Read the CANbus data stream*/

  while(TRUE)
  {
    {  
      unwind_protect_begin; // Catch errors
      do {
        ENTERBG; // This makes the CQ not block
        stat = canRead(hnd, &id, msg, &dlc, &flags, &time);
        LEAVEBG;
        // The message length dlc is always <= 8
        if (stat == canOK) // The communication channel was OK
        {
          if ((flags & canMSG_ERROR_FRAME) == 0)
          {  
            a_setintelem(tpl, 1, time, FALSE); // ts
            a_setintelem(tpl, 2, id, FALSE); // fr
            datap = (LONGINT *)msg;
            a_setintelem(tpl, 3, *datap, FALSE); // v
            //cast the Canbusdata to 64bit int
          }
        }
      }
    }
  }
}
```

---

**Turn off and Close:**
Before exit the wrapper, the CANbus channel should be turn off and close.
a_setintelem(tpl, 4, flags, FALSE); // fl
a_emit(cxt, tpl, FALSE); // emit the tuple
} else {
a_error(frame_error, mkinteger(id), FALSE);
}
} while (stat == canOK);

unwind_protect_catch; //all errors are caught
unwind_protect_end;

/*Turn off and Close */
canClose(hnd); // Close the channel
}

Two parameters of canreadbag() is tpl and cxt.cxt is an internal Amos II data
structure for managing the call and tpl is a tuple representing actual arguments and
results. The argument values are first, followed by the unassigned result values of
the function.[13]
The CAN Bus data stream wrapper is defined as a foreign function implemented in C
called canreadbag() that iteratively generates a bag of data stream elements as
tuples from a CAN Bus channel. It is the main function of process continues queries
of CANbus data stream.

3.2.3 canRead

**Definition:**
canStatus canRead( const int hnd, long * id, void * msg, unsigned
int * dlc, unsigned int * flag, unsigned long * time )

**Description:**
read CAN bus data from the receive buffer msg, hnd is handler of CAN channel,id is
the pointer of received CAN identifier (11 or 29), dlc is the pointer of receive
message length. flag receive the message flag, time receive the time stamp of
CANbus data.
Example:

canRead(hnd, &id, msg, &dlc, &flags, &time);

3.2.4 canOpenChannel

Definition:

```c
int canOpenChannel ( int channel, int flags )
```

Description:
Open a CAN bus channel and returns a handler calls to CANLIB. Channel's number is dependent on the CANbus device. The first channel always has number 0. In our case, the function will open can virtual device 0.

Example:

canOpenChannel(ctrl, canOPEN_ACCEPT_VIRTUAL);

3.2.5 canClose

Definition:

canStatus canClose ( const int hnd )

Description:

close the CANbus channel specified id.

Example:

canClose(0);

3.2.6 Error management for SCQS

Definition:

```c
int a_error(int errno, int obj, int catcherror)
```
**Description:**

The system also has the ability to raise errors to notify the Amos II system.

This function will signal a system error `errno` for the object `obj`. And `catcherror` is controller. FALSE means the system get an internal error jump and the control will not return to the caller. TRUE means the error exception raised after the origin function exited.

**Example:**

```c
a_error(frame_error, mkinteger(id), FALSE);
```

In this project, when an error frame is received, the system will set the id of the CAN bus data frame as an error no to AmosII.

---

**4 Running the CAN Wrapper**

The project will build a DLL file named `CANwrapper.dll`. User should put the DLL file in the bin folder of SCSQ so the system can load the foreign function implementation dynamically.

**4.1 Running steps**

Step1: Running the CanKing software. Enter the *traffic generator*. Set up the parameters of CAN Bus data frame in the CAN Bus Loader control panel. The important parameters are interval, burst size, count, id (11bits or 29 bits) and data. Click *Go On Bus* in the control panel to start the loader to produce data in the Virtual CAN Bus interface.

Step2: Save database image which support the foreign CAN bus related function on disk. Run commands `sace canwrapper.amosql`. The amosql file will load CAN bus data extension in C(a dll file), declare the CAN bus related foreign function, then save
database image to the disk.

Step 3: Run SCSQ with the saved image, `scsq canwrapper.dmp`.

Step 4: In the SCSQ system, query the CAN bus channel. `canreadbag(0)`

```
Release 16. v13
Amosql 1 ) canreadbag(0);
<63,830,35646,2>
<163,831,35647,2>
<263,832,35648,2>
<363,833,35649,2>
<463,834,35650,2>
<563,835,35651,2>
<663,836,35652,2>
<763,837,35653,2>
<863,838,35654,2>
<963,839,35655,2>
<1063,840,35656,2>
<1163,841,0,2>
<1263,842,0,2>
<1363,843,35659,2>
<1463,844,25659,2>
```

*Figure 4.1* CAN bus data stream

4.2 Process the CAN bus data by AMOSQL

To demonstrate AMOSQL can process CAN bus datastream, comparison is needed. First, setup the range of CAN bus id from 0-100. Then we can have two sets of tests, Test A(original data) and Test B(processed data).
The next step is create the AMOSQL function in the amosql file.

```
create function canreadrecords (Number Channel) -> Stream of Record
/* A function to create a stream of JSON records from Can Bus stream */
  as streamof(select {"ts": ts, "frame": fr, "value": v, "flags": fl}
               from Number ts, Number fr, Number v, Number fl
               where (ts, fr, v, fl) in canreadBag(channel));

set :r = canreadrecords(0);
/* Get the raw data stream of records */
in(:r);
```
Run an AMOSQL to process the CAN bus datastream where Id is greater than 80.

/* Run a stream of records on frame bigger than 80 */
select r from Record r where r in canreadrecords(0) and r["frame"] > 80;
5 Future work

This project implemented a CAN Bus wrapper for the SCSQ system. It is an extension enables SCSQ to process continuous queries of data from equipment producing data streams following the CAN Bus protocol. Using the CAN Bus wrapper, data from many different kinds of devices can be queried.

In future the system should be tested with more CAN Bus enabled devices like PCICAN, USBCAN, blackbirds, Eagle [12] and so on. Error management could be introduced so that when the system receives error frames indicating overheat lost connection, low power and so on, the system can signal an alarm or alert the user. Higher level CAN Bus data protocol can be implemented for industry usage.
References


[3]  https://www.kth.se/social/upload/64/CAN.pdf


[13]  Uppsala University, Department of Information Technology, Amos II C Interfaces, Tore Risch,  http://www.it.uu.se/research/group/udbl/amos/


[16]  http://www.it.uu.se/research/group/udbl/SCSQ.html