MultiCAN – EPICS goes CAN with Multiple Protocol Support

Jens Bergl, Ralph Lange, Götz Pfeiffer
BESSY GmbH
Lentzeallee 100
14195 Berlin
Germany

Table of Contents

Chapter 1: MultiCAN Design

The basic design of a modular multi-protocol CAN integration into EPICS was first presented at the ICALEPCS ’95 in Chicago[3].

Within the BESSY II control system the CAN field bus will serve a variety of tasks: In addition to the 'normal' device control it will interface underlying intelligent subsystems (like measurement PCs, PLCs), can be used for downloading configuration data and possibly object code to the embedded controllers, carries the ID compensation data and can be used for synchronization of IOC tasks. These different communication tasks require different protocols to be used in parallel on one CAN segment.

The CAN interface has to be implemented on three platforms: The IOC (Motorola MVME with VxWorks/EPICS and commercial 68000-based CAN cards), the embedded controller (i80386EX without OS with a single on-board CAN chip) and a PC used for measurement and debugging (80x86 using DOS and an AT bus CAN card).

SCI — A Hardware Independent Data Link

SCI, the *Simple CAN Interface*, is the specification of a library that embodies a standard for accessing CAN interface cards independent of the actual hardware[6]. Programs that use this library to access the CAN field bus will be portable between different CAN cards and even different operating systems. SCI is intentionally kept simple and it largely represents the data-link layer of the OSI model.

SCI is prepared to handle several CAN ports which are distinguished by port numbers. The ports may be realized on one or more interface cards, not necessarily from the same vendor. Different interface cards may have to be accessed through different drivers, but these differences are hidden by SCI.

SCI is capable of multithreading. In order to facilitate this function, each thread that opens the library is provided with a unique pointer. This pointer, which is a parameter for all SCI functions, is used to distinguish different threads. The properties of a

COB[1] — identifier, data length, timeout and type, where type can be one of read, write, remote-read and remote-write — are defined when it is initialized. SCI then returns a pointer to an internal structure that is allocated and initialized per object. All other functions use this sci_object pointer as a handle to a certain object. Read and write functions exist in order to transfer data to and from CAN objects.

Reading can be done in three basically different ways: *Immediate read* gets whatever data is stored on the CAN hardware for a certain object. *Read with timeout* gets new data or — if there is no new data — waits for the COB to arrive. *Install a callback function* that is called when new data arrives for an object is the asynchronous mode to retrieve data. On multitasking operating systems this callback is called from within a signal handler.

Design Overview

The requirements of portability and reusability are fulfilled best by a layered interface structure. This allows the parallel use of different protocols through well-defined interfaces into the protocol layer and collects the platform dependencies in another layer. Thus the support for a protocol can be implemented fully independent from hardware and OS properties. Figure 1 on page 7 shows the task level structure of the CAN interface in front of its layer structure.

Above the low level CAN library SCI is the GPS layer (*Generic Protocol Support*) hiding the Operating System and hardware dependencies. The tasks and functions in this layer set up the data transfer between the CAN segments (through SCI) and a COB oriented cache structure which is an interface to the next layer handling the different protocols. This layer consists of a small set of functions for each protocol to be run over the CAN bus. These functions include initialization, reading and writing data in protocol specific format (the upper interface) as well as reading and writing data in SCI conformal format (the lower interface).

EPICS device support writes and reads data to/from the cache using the upper interface functions of the appropriate Protocol Function Set. For output records the write request is fed into a queue. The mCANWriter dequeues it, fetches the data using the lower interface of the protocol layer and calls SCI to send the message. If SCI rejects the message (there might be minimal delays between messages to be kept), the Writer puts it back into the protocol, where it gets buffered and is resent after the SCI-requested delay. For asynchronous record processing an EPICS callback may be requested. On an incoming message SCI calls the mCANReader task. The mCANReader gets the COB, stores it through the appropriate protocol function and eventually requests an EPICS callback, which gets the data out of the protocol layer and moves it to the EPICS record. To minimize any mutual influences between protocols there is a mCANReader task for every protocol in use (not shown in figure). A third task, mCANTimer, periodically checks the protocol's status to detect timeouts caused by failures of the CAN hardware or the peer CAN node.

1. *Communication Object*, message with a unique tag.

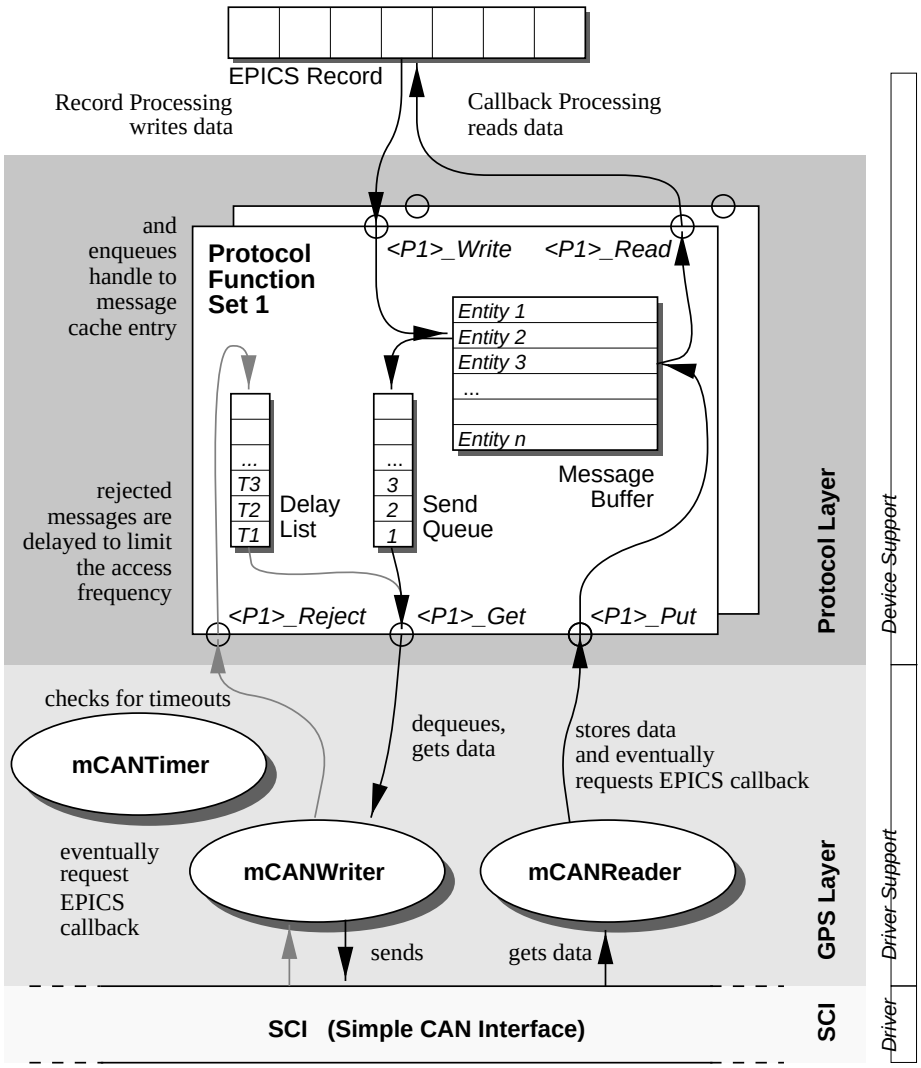

Figure 1: Overview of the MultiCAN Concept

Multiple Protocols

It is intended to implement the following protocols (i.e., their Protocol Function Sets):

- *LowCAL* is a subset of the CAL (*CAN Application Layer*)[1] protocol defined by CiA[1]. CAL is a huge protocol suite defining an open CAN environment, including services for message transfer, network management, on-line address distribution as well as remote configuration of certain layer parameters. Our lightweight LowCAL subset defines only the message specification as well as the encoding rules for the variable types used in BESSY II device control. This will be the protocol used for communication with the embedded controllers.
- *CANal* (CAN arbitrary length) specifies two protocol variants for sending data of arbitrary length such as tables, configuration data or object code: A multicast transfer which is not confirmed (*LowCANal*) and a point-to-point transfer which is confirmed and allows partial re-transfer of data (*HighCANal*).

1. CAN in Automation International Users and Manufacturers Group e.V.

Since the Protocol Function Sets are almost symmetrical and platform independent, the implementation of a new protocol is cut down to implementing its set of functions, which should be running on all target systems (i.e., the IOC as well as the embedded controller or PC).

Informations for using MultiCAN

Initialization

The use of the several protocols united in MultiCAN requires

- a configuration file *config.<machine-name>*
- the initialization of MultiCAN with mCANInit(char* *config_file*)
- the initialization of GPS described in “Initialization” on page 26
- the initialization of the protocols[1] as described in “Interface” on page 15

The configuration file *config.<machine-name>* has the following format:

<Number of ports>

<Port> <Bitrate>

These supported bitrates are :

- 1000 kbit
- 500 kbit
- 250 kbit
- 66 kbit

The use of MultiCAN by applications beside EPICS

If MultiCAN is used by EPICS, the whole initialization process is closely connected with EPICS. The way for other applications to use MultiCAN is the function

mcan_attach

mcan_attach (char* *protocol_name*, int *number_of_elements*, int (*) (void) *pointer_to_function*);

It allows an application to announce a number of entities on one of the supported protocols. The application has to provide a function, which is called in a later pass of the initialization by the device support of the corresponding protocol. The pointer of that function is the last parameter of mcan_attach(). That function must contain the calls for the initialization of all entities of that protocoll used by the application.

The detailed explaination where to call mcan_attach() can be found in “How to build it” on page 27.

1. At the moment only LowCAL is implemented

Chapter 2: Protocols

Common Lower (GPS) Interface

The interface between the different protocols and the GPS layer consists of the following functions:

<protocol>_get

prot_Return *<protocol>*_get (mcan_Obj_Handle* *obj_handle_p*, byte* *data_p*);

Get raw data from *protocol*.

obj_handle_p	Returns the object handle for which data is to be sent	
data_p	Returns the raw data (max. 8 bytes) to send	
Return value	Flag set may consist of	
	PROT_ERR	Error occurred
	PROT_EMPTY	No message to be sent
	PROT_DO_PP	If set, GPS is requested to arrange post-processing after successful write
	PROT_RD_PP	If PROT_DO_PP is set, this flag specifies the type of post-processing: if set, read post-proc is to be executed, otherwise write post-proc
	PROT_INHIB	Non-zero inhibit time; requests the use of sci_write_inhibit instead of sci_write
	PROT_OK	No errors; i.e., valid object handle and data, no post-processing, no inhibit time

This function is called by the writing task of the GPS layer to get the object handle and the raw data of a message to be sent over the CAN bus.

<protocol>_reject

void *<protocol>*_reject (mcan_Obj_Handle *obj_handle*, unsigned long *delay*);

Return message to be delayed back to *protocol.*

obj_handle	Object handle of message to be delayed
delay	Demanded delay [in µs]

GPS uses this function to return those messages back to *protocol* that were rejected by sci_write_inhibit because of their inhibit time not having expired. The protocol takes the message back and organizes re-notification of the GPS writer task after *delay* microseconds.

Caution: This function is defined only if inhibit time handling is enabled (i.e., if MCAN_USE_INHIBIT is defined).

<protocol>_put

prot_Return *<protocol>*_put (mcan_Obj_Handle* *obj_handle_p*, byte* *data_p*, int *s_ret*);

Put incoming CAN message into *protocol*.

obj_handle_p	Points to where GPS keeps the object handle (may be changed by the protocol)
data_p	Pointer to raw data
s_ret	SCI flags returned on receiving this message
Return value	Flag set may consist of

PROT_ERR	Error occurred
PROT_DO_PP	If set, GPS is requested to arrange post-processing after successful write
PROT_RD_PP	If PROT_DO_PP is set, this flag specifies the type of post-processing: if set, read post-proc is to be executed, otherwise write post-proc
PROT_OK	No errors; i.e., valid object handle and data, no post-processing, no inhibit time

This function is called by the reading task of the GPS layer to put the object handle and the raw data of a received message into the protocol layer. The object handle may be changed by the protocol, if post-processing is desired for a different object. This may be needed for different variables in a LowCAL variable set, where one object receives the data for all variables of a variable set.

<protocol>_time

prot_Return *<protocol>*_time (mcan_Obj_Handle* *obj_handle_p*);

Check for timeouts.

obj_handle_p	Returns the object handle for which timeout post-processing is demanded
Return value	Flag set may consist of

PROT_PEND	More timeouts may be pending — *<protocol>*_time has to be called again
PROT_DO_PP	If set, GPS is requested to arrange post-processing
PROT_RD_PP	If PROT_DO_PP is set, this flag specifies the type of post-processing: if set, read post-proc is to be executed, otherwise write post-proc
PROT_OK	No errors; i.e., valid object handle and data, no post-processing, no inhibit time

This function is called by the GPS timeout watchdog task in regular intervals (of TIMEOUT_RESOLUTION, defined in *multican.h)*.

LowCAL

Definition

The CAN Application Layer (CAL) provides four application layer service elements:

1. CAN based Message Specification (CMS)
   CMS offers an open, object oriented environment to design user applications. It specifies Variable, Event and Domain objects, their data types, the CAN access protocol and the encoding rules.
2. Network Management (NMT)
   NMT offers an environment to let one module (the NMT Master) deal with the initialization and possible failures of the other modules (NMT Slaves).
3. Distributor (DBT)
   The DBT offers a dynamic distribution of CAN message identifiers (COB-IDs) by one module (the DBT Master) to the other modules (DBT Slaves).
4. Layer Management (LMT)
   LMT offers the possibility to let one module (the LMT Master) control the settings of certain layer parameters at another module (LMT Slave).

The architecture of CAL allows to implement and use these services independently. Since only small parts of the CAL definition are relevant for use in the BESSY II controls, this feature was used to minimize the implementation effort.

LowCAL is a minimal subset of the CAL specification's CMS service. It provides asynchronous transmission of variables.

Data Types

The following basic data types for variables are defined:

INTEGER (valid lengths are 8, 16, 32 bits)

UNSIGNED (valid lengths are 8, 16, 32 bits)

Additionally arrays of variables of these basic types can be defined, as long as the CMS Encoding Rules are obeyed.[1]

Attributes

The following attributes for variables are defined:

user_type:	One of [CLIENT, SERVER].
data_type:	See "Data Types" on page 11.
class:	One of the values [BASIC, MULTIPLEXED].
access_type:	One of the values [READ_ONLY, WRITE_ONLY, READ_WRITE].
inhibit_time:	$n * 100$ µsec, n 0

1. Usually the total length of the array must not exceed 7 bytes for Multiplexed, 8 bytes for Basic variables (exept Read/Write access types).

Usage

The server of a variable is the CAN node where the variable originally resides. Accesses on the server side do a local update/read of the variable's value, accesses from the client side lead to request and response messages being sent over the CAN bus.

Variable sets can be used to "multiplex" several variables. All these multiplexed variables will then be mapped onto the COBs that are used by that variable set. This indirect addressing scheme reduces the number of used COBs. Within a variable set the variables are identified by a unique 7 bit "multiplexor", so the number of variables in a set is limited to 128. Since the multiplexor is transmitted in the data part of the CAN message, the maximum data size for a multiplexed variable is reduced to 7 bytes.

The access type of a variable is seen from the point of view of the client. Read-only variables can be used by a client only to collect data. The collected data will be the data that was set by the server in its last write access — previous updates are lost. Write-only variables can be used by a client to request one or more servers to execute a command. This access is not confirmed (multicast). Read-write variables can be used by a client to collect the 'current data' from the server (read access) or to request the server to take a value and/or execute a command (write access). Both read and write accesses are confirmed.

The inhibit time enforces a minimal time delay between accesses on the same variable resp. variable set. This can be used to slow down a client not to exceed the capabilities of the variable's server.

CAN Protocol

Order and type of the messages sent over the CAN bus during a LowCAL variable access strongly depend on the attributes of that variable. There are unconfirmed and confirmed services ('service' being a read or write variable access). The unconfirmed services use one COB that may have more than one recipient, confirmed services usually use two COBs (one for each direction) and connect exactly one server/client pair.

Basic Variables, Read Only Access

One confirmed service (Read Variable) is defined. The client sends a *Remote Transmit Request* (RTR) frame over the CAN, the server responds with a message containing the requested data. One COB is needed for the connection. There has to be exactly one server and an arbitrary number of clients.

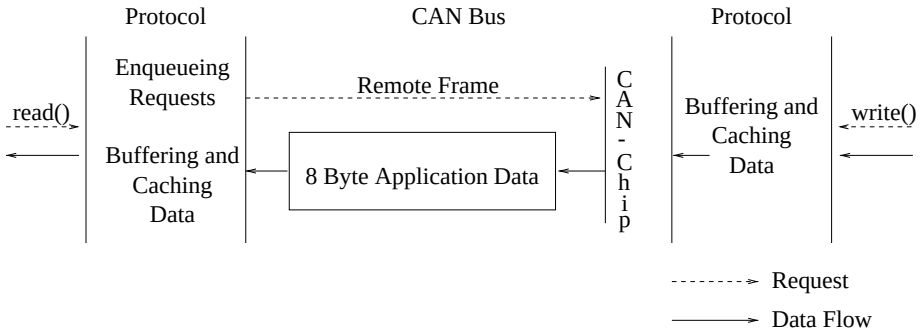

Figure 2: CAN Protocol for Basic Variables, Read Only Access

Application Data: up to 8 bytes of data representing a value of the data type attribute of the basic variable.

Multiplexed Variables, Read Only Access

One confirmed service (Read Variable) is defined. The client sends a request frame which contains the multiplexor of the variable to be read. The server returns the data and a flag indicating the result of the operation. Two COBs are needed for the connection (one for each direction). There has to be at most one server and one client per connection.

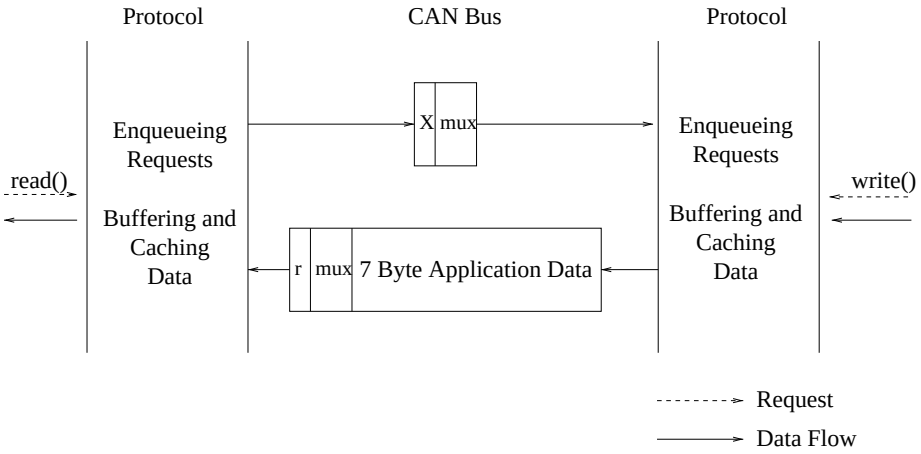

Figure 3: CAN Protocol for Multiplexed Variables, Read Only Access

mux:	Multiplexor, a value between 0 and 127 (inclusive)	
X:	not used, always 0	
r:	result	
	0:	Success
	1:	Failure

Application Data: up to 7 bytes of data. If r = 0, it represents a value of the data type attribute of the multiplexed variable identified by mux. If r = 1, it represents a value of the error type attribute of this variable.

Basic Variables, Write Only Access

One unconfirmed service (Write Variable) is defined. The client sends a data frame to the server(s). One COB is needed for the connection. There may be one or more server(s) and at most one client.

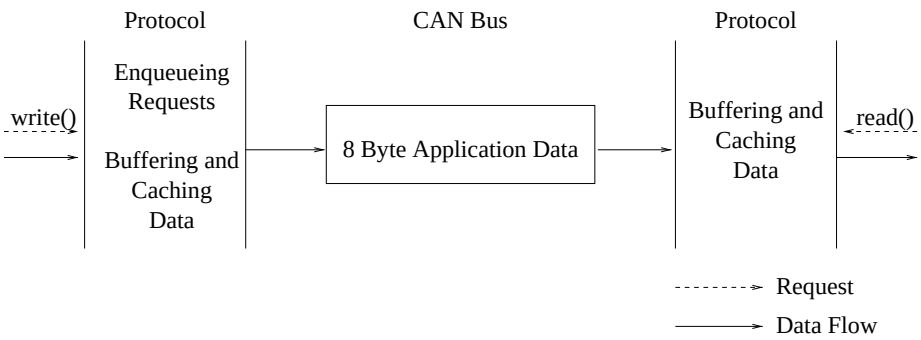

Figure 4: CAN Protocol for Basic Variables, Write Only Access

Application Data: up to 8 bytes of data representing a value of the data type attribute of the basic variable.

Multiplexed Variable, Write Only Access

One unconfirmed service (Write Variable) is defined. The client sends a data frame to the server(s). One COB is needed for the connection. There may be one or more server(s) and at most one client.most one client.most one client.

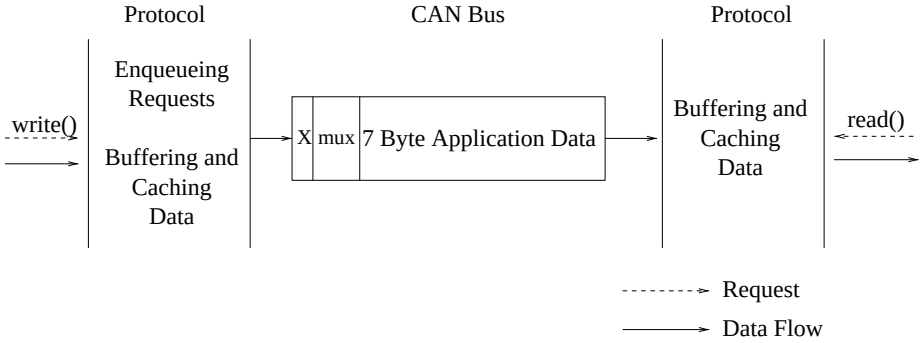

Figure 5: CAN Protocol for Multiplexed Variables, Write Only Access

X: not used, always 0

Application Data: up to 7 bytes of data representing a value of the data type attribute of the multiplexed variable identified by mux.

Basic and Multiplexed Variables, Read/Write Access

Two confirmed services (Read Variable and Write Variable) are defined. The client sends a request frame which contains the multiplexor of the variable to be accessed and a flag indicating the operation type. The server returns the data and a flag indicating the result of the operation. Two COBs are needed for the connection (one for each direction). There has to be at most one server and one client per connection.

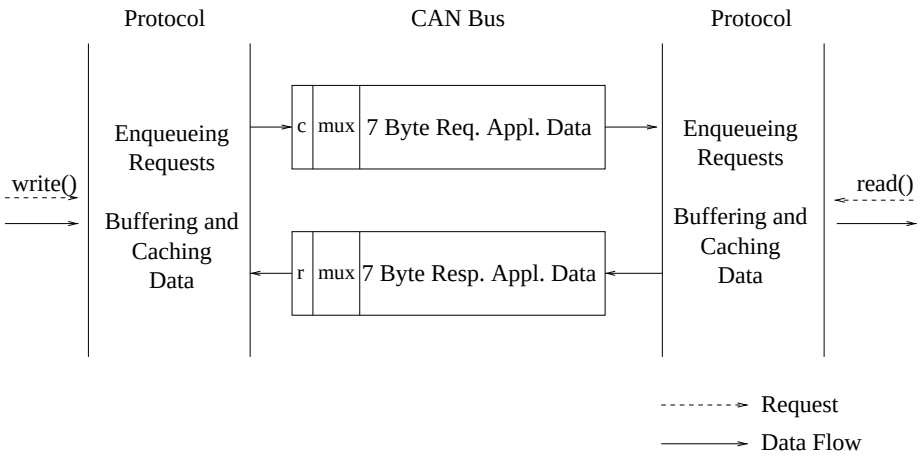

Figure 6: CAN Protocol for Basic and Multiplexed Variables, Read/ Write Access

mux: Multiplexor, only valid for multiplexed variables. If valid, a value between 0 and 127 (inclusive), otherwise 0

c: command specifier

- 0: Write
- 1: Read

Req. Appl. Data: only valid when c = 0, contains up to 7 bytes of data, representing a value of the data type attribute of the basic variable resp. the multiplexed variable identified by mux.

r: result

0: Success

1: Failure

Resp. Appl. Data: up to 7 bytes of data. In case of a write response and r = 0, it represents the same value as passed with the write request. In case of a read response and r = 0, it represents a value of the data type attribute of the multiplexed variable identified by mux. In case of r = 1, it represents a value of the error type attribute of the basic variable resp. the multiplexed variable identified by mux.

Encoding Rules

The CAN message encoding rules are conformant with the CMS (CAL) encoding rules[1]. Since LowCAL defines a subset of the CMS data types (only 8 bit aligned types), the encoding rules can be simplified as follows:

The transfer encoding syntax equals the representation syntax for little endian two's complement computers.

Figure 7 shows the encoding for an array of two signed short integer variables.

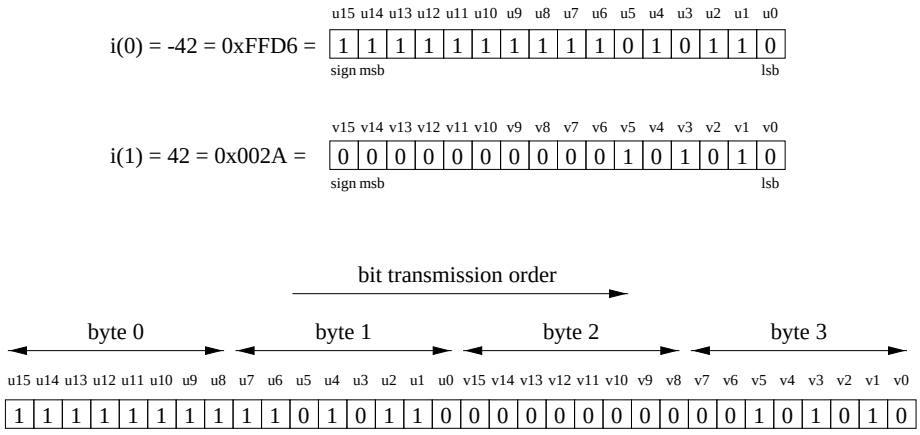

Figure 7: Encoding Rules (Example)

Implementation

Interface

LowCAL's interface (application side) consists of the following functions:

lowcal_init_start

prot_Return lowcal_init_start (unsigned short *el*);

First pass initialization routine.

el	Number of LowCAL entities (variables) that will be used
Return value	Flag set may consist of
PROT_ERROR	Error occurred
PROT_OK	No errors

This first pass of initialization allocates buffers, semaphores and queues. The SCI and GPS layers are initialized.

In a multi application environment, this function may be called only by the main application that starts up MultiCAN. Other applications have to use mcan_attach to announce their use of LowCAL.

lowcal_init_entity

```
prot_Return
lowcal_init_entity ( unsigned short port,
                     unsigned short out_id,
                     unsigned short in_id,
                     byte max_length,
                     mcan_Obj_Handle* obj_handle_p,
                     lcal_Attrib var_attrib,
                     lcal_Data_Type var_type,
                     unsigned long timeout);
```

Second pass initialization routine.

port	Number of the CAN segment to be used
out_id	COB ID for outgoing data
in_id	COB ID for incoming data
max_length	Maximum length of this variable's data For multiplexed variables: maximum length of this variable set's data (including multiplexor byte)
obj_handle_p	Returns an MultiCAN object handle to use in subsequent calls
var_attrib	LowCAL attribute setof the variable to be initialized
var_type	LowCAL data type of the variable to be initialized
timeout	Timeout value [ms] for confirmed services
Return value	Flag set may consist of

PROT_ERR	Error occurred
PROT_INIT	Variable was already initialized. The returned object handle is a duplicate.
PROT_OK	No errors

The second pass of initialization initializes a variable's buffer entry, creates a new variable set (if needed) and initializes the SCI in and out objects used for this variable. Arguments might be ignored, if not applicable (e.g., *in_obj* resp. *out_obj* and *timeout* for unconfirmed services). The returned object handle is used to identify the variable in subsequent LowCAL calls.

lowcal_init_end

```
prot_Return lowcal_init_end (void);
```

Third pass initialization routine.

Return value	Flag set is always

PROT_OK	No errors

This routine marks the end of LowCAL initialization — subsequent calls to lowcal_init_entity are erroneous.

lowcal_write

```
prot_Return lowcal_write (mcan_Obj_Handle obj_handle, void* data_p);
```

Write data to the protocol buffer and set up sending.

obj_handle	Object handle of the variable to use (as returned on lowcal_init_entity call)
data_p	Pointer to data to be written (of the type declared in its lowcal_init_entity call)
Return value	Flag set may consist of

PROT_ERR	Error occurred
PROT_TIMO	Only returned on a client's second call to lowcal_write: A timeout occurred (i.e., post-processing was initiated by the timeout watchdog task, not the confirmation of the write access)

PROT_FAIL	Only returned on a client's second call to lowcal_write: A failure occurred (i.e., the confirmation of the last write access had its failure bit set)
PROT_OK	No errors

On the client side, this routine copies the data into the protocol's cache buffer. If there is no request pending, it issues the sending of the appropriate write request by putting the request into the send queue and notifying the writer task.

On the server side, this call copies the data into the protocol's cache buffer and sets the variable's buffer entry valid.

lowcal_read

prot_Return lowcal_read (mcan_Obj_Handle *obj_handle*, void* *data_p*);

Read data from the protocol buffer, eventually set up sending of a read request (client side only).

obj_handle	Object handle of the variable to use (as returned on lowcal_init_entity call)
data_p	Pointer to data to be read (of the type declared in its lowcal_init_entity call)
Return value	Flag set may consist of

PROT_ERR	Error occurred
PROT_TIMO	Only returned by a client's second call to lowcal_read: A timeout occurred (i.e., post-processing was initiated by the timeout watchdog task, not the confirmation of the read access)
PROT_FAIL	Only returned by a client's second call to lowcal_read: A failure occurred (i.e., the confirmation of the last read access had its failure bit set)
PROT_OLDAT	Only returned on a client's call to lowcal_read: The data copied to data_p is old (i.e., it has been read already)
PROT_OK	No errors

On the client side, this routine copies the cache buffer's contents to the variable *data_p* points to. If there is no request pending, it issues the sending of the appropriate read request by putting the request into the send queue and notifying the writer task.

On the server side, this call copies the cache buffer's contents to the variable *data_p* points to.

lowcal_pp_complete

void lowcal_pp_complete (mcan_Obj_Handle *obj_handle*);

Signal the completion of post-processing.

obj_handle	Object handle of the variable to use (as returned on lowcal_init_entity call)

Calling this function signals the completion of post processing to the protocol. The buffer entry's flags are set accordingly. On the client side, if a read or write access has been postponed because of post-processing being busy, it is issued by putting the request into the send queue and notifying the writer task.

lowcal_get_inobj

sci_Object* lowcal_get_inobj (mcan_Obj_Handle *obj_handle*);

Get pointer to SCI object for incoming data.

obj_handle	MultiCAN object handle (variable) to use
Return value	Pointer to the SCI object used for a variable's incoming data

Caution: This function is defined only if MultiCAN runs on one processor (i.e., if MCAN_DISTRIBUTED is not defined).

lowcal_get_outobj

sci_Object* lowcal_get_outobj (mcan_Obj_Handle *obj_handle*);

Get pointer to SCI object for outgoing data.

obj_handle	MultiCAN object handle (variable) to use
Return value	Pointer to the SCI object used for a variable's outgoing data

Caution: This function is defined only if MultiCAN runs on one processor (i.e., if MCAN_DISTRIBUTED is not defined).

Chapter 4: EPICS Device Support

Using MultiCAN for transmitting data from an EPICS database to a variety of devices over the CAN field bus needs appropriate device support. The runtime interface of that layered architecture is provided by the set of protocols. Every implemented protocol needs an own device support. At the moment the protocol LowCAL is supported for the following record types:

- String Input and String Output record
- Analog In and Analog Out record
- MultiBit Binary In Direct and MultiBit Binary Out Direct record

The several steps of the initialization of MultiCAN are closely connected with the initialization of EPICS.

The properties of the communication channel and the peer entity of the EPICS record are specified in the hardware link.

Usage of MultiCAN with EPICS

How to build it

The first step is the building of the MultiCAN core *mCANCore.* Now the environment variable *MCAN* has to be set to the absolute path of the subdirectory of MultiCAN.

The following modules have to be rebuilt[1] for supporting MultiCAN:

- *drvSup* in *.../epics/base/src/drv*
- *devSup* in *.../epics/base/src/dev*
- *initHooks.o*

The new building of the module *drvSup* with set variable *MCAN* causes a compiling of d*rvmCAN.c* and the linking of *drvmCAN.o* and *mCANCore* to *drvSup.*

1. *Rebuild* means to do a *gmake* in the corresponding directory.

The new *devSup* results from the compilation of the device support functions of the several protocols of MultiCAN and the linking of it to *devSup.*

For initialization reasons the file initHooks.c has to be compiled after the setting of *MCAN*. If it is intended to use MultiCAN with other applications beside EPICS, the environment variable *ATTACH* must be set. Besides the file *.../epics/base/src/db/initHooks.c* has to be modified in the here shown way:

1. Insert in the *USERAREA-2* the call of
   mcan_attach(char* protocol_name, int number_of_elements, int (*) (void) pointer_to_function);
   Example:
   ```
   /*###BEGIN OF USERAREA-2###*/
   mcan_attach("lowcal",1,test_init1_attach)
   /*###END OF USERAREA-2###*/
   ```
2. Insert the declaration of the extern user function, which has to do the initialization of the entities of the corresponding protocol used by that application beside EPICS.
   Example:
   ```
   /*###BEGIN OF USERAREA-1###*/
    extern int test_init1_attach(void);
   /*###END OF USERAREA-1###*/
   ```
3. New compilation of *initHooks.c*

More detailed information about attaching applications to the MultiCAN initialization done by EPICS can be found in “The use of MultiCAN by applications beside EPICS” on page 8.

How to use it

The use of the CAN fieldbus with MultiCAN by EPICS requires the correct setting of two fields of the corresponding records.

The appliance of the protocol LowCAL claims the setting of the field **DTYP** to **lowcal.**

The other is field is the **INP** field or the **OUT** field. It has to contain a string with the maximum length of 35 characters**.** That sequence of characters describes the connection to another entity using MultiCAN.

For LowCAL that string has the following syntax. Like for all strings in these fields the first character is’**@**’. It is followed by the first character describing attributes of the LowCAL variable. These are the meanings of the several permissible characters:

Character	*User type*	*Class*	*Access type*
a	Client	Basic	Read-only
b	Client	Basic	Write-only
c	Client	Basic	Read-Write
d	Client	Multiplexed	Read-only
e	Client	Multiplexed	Write-only
f	Client	Multiplexed	Read-Write
g	Server	Basic	Read-only
h	Server	Basic	Write-only
i	Server	Basic	Read-Write

j	Server	Multiplexed	Read-only
k	Server	Multiplexed	Write-only
l	Server	Multiplexed	Read-Write

The next character, separated with *<space>* from the first, specify the data type of the LowCAL variable as shown in the next table

Character	LowCAL Data Type
a	String
b	String being encoded or decoded by the device support[a]
s	Signed Short
S	Unsigned Short
t	Array of Signed Short
T	Array of Unsigned Short
u	Raw Signed Short
U	Raw Unsigned Short
v	Array of Raw Signed Short
V	Array of Raw Unsigned Short
l	Signed Long
L	Unsigned Long
m	Array of Signed Long
M	Array of Unsigned Long
n	Raw Signed Long
N	Raw Unsigned Long
o	Array of Raw Signed Long
O	Array of Raw Unsigned Long
c	Signed Character
C	Unsigned Character
d	Array of Signed Character
D	Array of Unsigned Character

a.Encoding or decoding means the replacement of '\0' in incoming strings before writing it to the record respectively regenerating of replaced '\0' in outgoing strings. More information about the use of it can be found in "Special Features" on page 30.

The different arrays of long can only be used with basic variables because 8 is the maximum number of bytes you can transmit with one CAN data frame.
The succeeding 10 hexadecimal integers are separated with **<space>** and have the meaning as shown in that table:

1. **Maximum length** of data in byte to transmit with that object id (COB) on the CAN bus. If you use multiplexed variables, one byte for the multiplexor is to be

added to the size of the chosen data type. All multiplexed variables using the same CAN object id (COB) must have the same maximum length.

2. The **port** number on the CAN card depending on the jumper settings of that card
3. CAN **object id** (COB) of the **outgoing** object
4. CAN **object id** (COB) of the **incoming** object
5. The **multiplexor**
6. The **inhibit time** is the minimum time to elapse between the sending of CAN objects with the same CAN object id. The time t results from the input value multiplied with a constant of 100µs.
7. The **timeout value** refers to confirmed or remote LowCAL services[1] and can be adjusted in a resolution of 1ms.
8. The meaning of that integer depends on the record type and the LowCAL data type. For mbbiDirect and mbboDirect record it represents a shift value which is explained in [2]. If the LowCAL data type is an array it represents the size of the array.

Not for all LowCAL variable types all parameters are necessary but the device support for LowCAL expects all above mentioned entries in the string. Not needed parameters should be set to ’0’.

Here is an example of an hardware link to LowCAL with the following attributes:

- Client / Multiplexed / Read-Write
- Multiplexor 10
- Unsigned Short
- Maximum length of 3
- Port 2 / Out-id 257 / In-id 193
- Inhibit time of 500µs
- Timeout value of 500ms

```
"@f S 3 2 101 c1 a 5 1f4 0"
```

Special Features

At the moment there are no array records, for instance array of short, available in EPICS. For reading and writing such arrays to or from a set of single records over the CAN bus it needs a workaround. With the help of a string record and a sequencer it is now possible to substitute the missing record types. The in [4] described sequencer realizes the connection between the single records and the string record. The string record is connected over its hardlink via MultiCAN to a peer entity using LowCAL too. For avoiding an occurrence of ’\0’ in the string, what would lead to an arbitrary cut off, encoding and decoding functions are used in the sequencer and in the device support. The setting of the data type in the hardware link to’b’ activate that feature. Every simulation of an array record requires one separate state set in the state program.

1. All LowCAL clients of the access types Read-only and Read-Write.

References

[1] *CAN Application Layer.* CiA/DS201 ... CiA/DS205, CiA/DS207. May 1995. CiA e.V., Simon-Schöffel-Str 21, D-90427 Nürnberg.

[2] J. Anderson, M. Kraimer: *EPICS Input / Output Controller (IOC) Record Reference Manual.* Argonne National Laboratory – Advanced Photon Source. November 1994.

[3] J. Bergl, B. Kuner, R. Lange, I. Müller, R. Müller, G. Pfeiffer, J. Rahn, H. Rüdiger: *Controller Area Network (CAN) — a Field Bus Gives Access to the Bulk of BESSY II Devices.* Presented at the ICALEPCS ’95, Chicago, 1995.

[4] A. Kozubal: *State Notation Language and Run-time Sequencer Users Guide*. Argonne National Laboratory – Advanced Photon Source. September 1993.

[5] M. Kraimer: *EPICS Input / Output Controller (IOC) Application Developer’s Guide*. Argonne National Laboratory – Advanced Photon Source. November 1994.

[6] B. Kuner, R. Lange, I. Müller, G. Pfeiffer, J. Rahn: *SCI — Simple CAN Interface.* BESSY, Berlin, 1996.

Index

T

U

V