Communications API

TEAM A : Communications and Integration Group

April 15, 1995

1 Introduction

This document specifies the API provided by the Communications and Integration group for use in the AMC system. This document will outline the purpose of the library, its contents and availability and then a description of the functions that can be used.

While primarily for ROBOT and VISION, this may be of interest to the SCADA as a reference implementation of the Communications and Integration Group's specifications. Where the functionality in this library differs from that specified in the standards, the standards prevail; however the Communications and Integration Group should be notified of the problem so that they can rectify the inconsistency.

2 Purpose

This API has been provided for several reasons:

1. It provides one single reference implementation for the communications specification which can not only be used by subsystems, but can be used to verify other implementations (a la "bakeoff" style).
2. It means subsystems do not have to deal with possibly unfamiliar communications concepts in their own implementation of the communications specification(s).
3. It minimizes testing and debugging: by using this API, it removes the need to test each particular subsystems implementation of low level communications including message syntax (effectively means that message semantics are all that are tested).

3 Related Specifications and Documents

1. Communications Specification
2. Revision Control Specification

4 API components

The API is provided as a directory of source files, header files, and a single library. There are only two files that are used by users of the API, these are `COMMSAPI.LIB` and `COMMSAPI.H`. The source files are provided as they may help in the understanding of the library. You should not recompile and/or use "internal" components of the API other than what has been specified in this document. The Communications and Integration Group will take no responsibility for the consequences.

You should ensure that your current version of the library is the most up to date one. There are two facilities for doing. Each component within the library has an RCS version control string which indicates its revision. The header files also have this. You can use the RCS 'ident' program to check these against what has been specified as the current revision set. At the time of writing this document, the following are the current revisions:

```
d:\uts\a95\csd\comms\src# ident commsapi.lib
commsapi.lib:
     $Id: serial.c 1.2 1995/04/15 03:51:15 teama-ci Exp $
     $Id: frame.c 1.2 1995/04/15 03:51:15 teama-ci Exp $
     $Id: messages.c 1.3 1995/04/15 03:55:59 teama-ci Exp $
     $Id: commsapi.c 1.1 1995/04/15 01:57:43 teama-ci Exp $

d:\uts\a95\csd\comms\src# ident *.h
COMMSAPI.H:
     $Id: commsapi.h 1.1 1995/04/15 01:57:43 teama-ci Exp $
FRAME.H:
     $Id: frame.h 1.2 1995/04/15 03:51:15 teama-ci Exp $
MESSAGES.H:
     $Id: messages.h 1.4 1995/04/15 03:55:59 teama-ci Exp $
SERIAL.H:
     $Id: serial.h 1.1 1995/04/15 01:57:43 teama-ci Exp $
SYSTEM.H:
     $Id: system.h 1.1 1995/04/15 01:57:43 teama-ci Exp $
```

You can obtain ASCII strings of the current source code revisions by using the functions `serial_version ()`, `frame_version ()` or `msg_version ()`. Alternatively, the `commsapi_version ()` will print out to standard output all the versions of internal components.

For example:

```
d:\uts\a95\csd\comms\src# type showver.c

#include <stdio.h>
#include "commsapi.h"

static const char rcs_id[] =
            "$Id: showver.c 1.2 1995/04/15 03:51:15 teama-ci Exp $";

int
```

```
main (int argc, char ** argv)
  {
    commsapi_init ();
    printf ("[%s]\n", rcs_id);
    commsapi_version ();
    return (0);
  }

d:\uts\a95\csd\comms\src# showver
[$Id: showver.c 1.2 1995/04/15 03:51:15 teama-ci Exp $]
COMMSAPI: $Id: commsapi.c 1.1 1995/04/15 01:57:43 teama-ci Exp $
  SERIAL: $Id: serial.c 1.2 1995/04/15 03:51:15 teama-ci Exp $
  FRAME : $Id: frame.c 1.2 1995/04/15 03:51:15 teama-ci Exp $
  MSG   : $Id: messages.c 1.3 1995/04/15 03:55:59 teama-ci Exp $
```

Using the API

The API is currently compiled for the SMALL memory model under Borland/Turbo C. Should you need a different model, please contact the Communications and Integration Group who will construct one.

There are only files that you need to use. The first is the library itself that contains the compiled API functions. The second is the header file which provides the defines, data structures and prototypes for the API.

1. Ensure that your compiler is set up to look for library and include files in the directory where the API is installed. You can do this by accessing the "Options" and then "Directories" menus. You should be able to edit the "Includes" and "Libraries" directories. Do not simply replace the existing directories; append to them by using a semicolon as the delimiter. E.g. `C:/TURBOC/INCLUDE` would then become `C:/TURBOC/INCLUDE;D:/TEAMA/COMMS/SRC`.
2. Ensure that you include the API library file in with your compilations. If you are using "Projects", then add `COMMSAPI.LIB` to your list of Project files. The API library will then be included during linking.
3. Ensure that all the 'C' files you use include `COMMSAPI.H`. The placement of the include does not matter, as any system includes that it depends on will be included by itself.
4. Before you use any of the API library functions, make sure that you initialise the API library by calling the `commsapi_init ()` function. This function takes no arguments and returns void. You should probably do this as one of the very first things in your software, and you should only do it once.

5 API Functions

There are two core sets of functions in the API. The first relates to communications services; i.e. the provision of an ability to establish a communications association with a specified peer. The second relates to message encoding and decoding. These are specified in [XXX].

5.1 Communications

The communications services provide a rudimentary reliable communications layer for use over asynchronous serial ports (8250 family, as used in the IBM PC and compatibles). Both levels of this communications paradigm are specified in [ref to spec] where, in OSI parlance, the "physical layer" corresponds to the RS232C layer and the "data link layer" refers to the overlaid system of framing and acknowledgments.

The implementation consists of interrupt driven 8250 specific code. When outgoing frames are sent, the transmitter will wait until it receives an acknowledgment from the peer. This is usually fairly quick, for the reason that the receiver implementation will reassemble frames under interrupt and also generate and send acknowledgments under interrupt. This is the only assumption in the implementation: that sending data back out to the transmitter which in an interrupt service routine (but disabling subsequent interrupts) is possible. Testing has shown that it is.

The messages sent with these services must satisfy two main preconditions (in addition to satisfying the syntactic rules defined later):

1. They must be null terminated strings, or the string must be a null itself (in which case the remote peer will receive the message but not provide it to the peer application: this serves as a non-intrusive "ping" facility).
2. They must be of a size less than `FRAME_MSG_SZ`. This is a define that is set in one of the header files included in `COMMSAPI.H`. The actual value of this is just less than 128 bytes, and its exact value will not be specified here as it is subject to change (you should always use the symbolic name).

The functions are:

Function:

```
frame_ctx *
frame_open (uchar s_port, uchar s_irq, uchar src, uchar dst, uint timeout, uint
retries)
```

Description:

This initialises a peer to peer association to be used for communications purposes. It sets up low level serial port routines and resets all state information relevant to the association.

Arguments:

`s_port` specifies the serial communications port on the PC to be used. This must be one of COM1, COM2, COM3 or COM4.

`s_irq` specifies the serial communications irq to be used on the PC. This can be, but is not limited to, COM1IRQ, COM2IRQ, COM3IRQ, COM4IRQ.

`src` specifies the identifier to be used for the source of the association (i.e. the `local machine name`, for example 'V' for VISION or 'R' for ROBOT).

`dst` specifies the identifier to be used for the remote peer on the association (i.e. the remote machine name).

`timeout` specifies the timeouts between retransmission of frames in MILLISECONDS.

`retries` specifies the number of retries that will be attempted, in between timeouts, before aborting a transmission.

Return:

The return is a pointer that is to be used as the unique identifier for this association. It is used as the argument in subsequent function calls using this module, and has data fields that can be accessed to provide state and statistical information. The `FRAME.H` header file defines this `frame_ctx` structure.

Function:

```
void
frame_reset (frame_ctx * ctx)
```

Description:

Resets all the state information for a given association, the purpose of this is to synchronise an association back to a known state. Within the AMC, it is used upon entry to the *uninitialised* state or after a communications error has occurred.

Arguments:

`ctx` specifies the association.

Return:

none.

Function:

```
void
frame_close (frame_ctx * ctx)
```

Description:

Closes down an association that was previously opened using `frame_open ()`. The association is now invalid and cannot be used (without a new call to `frame_open ()`). The association identifier is now invalid.

Arguments:

`ctx` specifies the association.

Return:

none.

Function:

```
bool
frame_send (frame_ctx * ctx, uchar * message)
```

Description:

This function sends (or attempts to send) a message to the remote peer in the association. The message is framed and transmitted over to the peer, and an acknowledgment is expected. If a transmission error occurs (of any kind), then the return will indicate so, and the error field for the association will carry the specific error reason.

Arguments:

`ctx` specifies the association. `message` specifies the null terminated string to be sent to the peer. This string may also be null itself (i.e. a null pointer).

Return:
`boolean` TRUE if transmission was successful, FALSE if transmission failed for any reason, with `error` set to indicate the reason.

Function:
```
uint
frame_msg_count (frame_ctx * ctx)
```

Description:
Indicates the number of messages queued for the application.

Arguments:
`ctx` specifies the association.

Return:
`unsigned integer` indicating the number of messages queued.

Function:
```
bool
frame_msg_getnext (frame_ctx * ctx, uchar * data, uint data_sz)
```

Description:
Obtains the next message (in FIFO form) that is queued for the application. The message is copied from the queue and, if required, is truncated to fit the buffer supplied by the application. An indication of whether a message was copied is returned. The message is deleted from the queue.

Arguments:
`ctx` specifies the association. `data` specifies the character array into which the message is to be copied. The message when copied will be null terminated. `data_sz` specifies the maximum size of the character array and is used to ensure that data corruption doesn't occur because of copying long messages into short(er) buffers.

Return:
`boolean` TRUE if a message was copied and is now in `data`, FALSE if one was not copied (i.e. there were none in the queue available).

Function:
```
bool
frame_msg_delnext (frame_ctx * ctx)
```

Description:
Deletes the next message in the queue if one is there to delete. This is an alternative to using `frame_msg_getnext ()` and simply throwing away the copied message.

Arguments:
`ctx` specifies the association.

Return:
`boolean` TRUE if a message was deleted, FALSE if a message was not deleted (i.e. there were none in the queue).

Function:

```
void
frame_msg_clear (frame_ctx * ctx)
```

Description:

Deletes all messages that are queued for the application.

Arguments:

`ctx` specifies the association.

Return:

none.

The contents of the `frame_ctx` data structure are:

```
typedef struct
  {
    bool            active;              /* is the association active ? */
    uchar           src;                  /* message src (ours usually) */
    uchar           dst;                        /* message dst (theirs) */
    uint            timeout;             /* timeouts between each retry */
    uint            retries;      /* number of retries before giving up */
    bool            overflow;                /* were there overflows ? */
    uchar           tx_seq;                      /* tx sequence number */
    uchar           rx_seq;                      /* rx sequence number */
    frame_tx_state  tx_state;                              /* tx state */
    frame_rx_state  rx_state;                              /* rx state */
    frame_error     error;                                    /* error */
    uchar           rx_frame[FRAME_SZ];                    /* incoming */
    uint            rx_frame_sz;                               /* size */
    uint            rx_error_frames;                /* errors in frames */
    uint            rx_ok_frames;                         /* ok frames */
    uchar           buffer[FRAME_BUFFER_SZ * FRAME_MSG_SZ];  /* storage */
    uint            buffer_head;                      /* head of buffer */
    uint            buffer_sz;                        /* size of buffer */
    serial_ctx *    sctx;             /* lower layer serial association */
  } frame_ctx;
```

where:

```
typedef enum
  {
    st_tx_idle,                                    /* idle; no activity */
    st_tx_waiting_for_ack,           /* waiting for ack back from peer */
    st_tx_received_ack,                               /* got the ack! */
    st_tx_error                                   /* really bad error ! */
  } frame_tx_state;

typedef enum
```

```
  {
    st_rx_looking_for_header,            /* idle: looking for a header */
    st_rx_looking_for_trailer,         /* got head, looking for trailer */
    st_rx_error                          /* unrecoverable error state */
  } frame_rx_state;

typedef enum
  {
    err_none,                                              /* no error */
    err_serial_tx_error,                            /* transmit failed */
    err_ack_timeout                       /* no acknowledgment received */
  } frame_error;
```

Example:

The following is an example of a rudimentary system that accepts and processes messages. The structure should not be construed as being that which should be used for the AMC system.

```
bool
system ()
  {
    frame_ctx * ctx;
    uchar in_message[FRAME_MSG_SZ+1];
    uchar out_message[FRAME_MSG_SZ+1];

    commsapi_init ();

    /* we are V and they are S, use COM2 with 750ms timeouts with
     * 3 retries
     */
    ctx = frame_open (COM2, COM2IRQ, 'V', 'S', 750, 3);
    if (ctx == NULL)
        return (FALSE);

    do
      {
        while (ctx->error != err_none)
          {
            /* comms error !
             */
            frame_reset (ctx);

            /* just keep trying until peer is reachable
             */
            frame_send (ctx, NULL);
          }

        /* loop indefinite until messages are available
         */
        while (frame_msg_count (ctx) == 0)
            ;
```

```
        /* get the message, if there is no message then
         * return back upwards
         */
        if (frame_msg_getnext (ctx, in_message,
                sizeof (in_message)) == FALSE)
            continue;

        /* ... process the contents of the message ... */

        /* send an output message, note that the error (if any) from
         * frame_send will be picked up when the loop is restarted
         */
        if (strlen (out_message) > 0)
          {
            frame_send (ctx, out_message);
          }

      } while (1);

    /* close the association
     */
    frame_close (ctx);
    return (TRUE);
  }
```

5.2 Message Encoding/Decoding

These routines provide simplistic functions for manipulating messages in the format specified in [xxx]. In particular, it is possible to construct messages, and it is possible to sequentially extract fields from messages.

The following system messages are already defined in the `MESSAGES.H` file that is included by `COMMSAPI.H`. You should structure your own subsystem specific messages using the same format (i.e. the two byte identifiers are encoded into a single word and macros are provided for achieving this purpose).

```
#define MSG_INITIALISE_REQ      MSGTYPE ('A','R')
#define MSG_START_REQ           MSGTYPE ('B','R')
#define MSG_STOP_REQ            MSGTYPE ('C','R')
#define MSG_SHUTDOWN_REQ        MSGTYPE ('D','R')
#define MSG_EMERGENCYSTOP_REQ   MSGTYPE ('E','R')
#define MSG_STATUS_IND          MSGTYPE ('F','I')
```

The functions provided are:

Function:
`uint`

`msg_get_type (uchar * message)`

Description:
Given a fully specified message, this function will return the message identifier associated with that message. It also initialises internal state information so that subsequent actions to extract fields will refer to this message. A copy of the message is kept internally.

Arguments:
`message` is the null terminated string which contains the message, in the format specified in [xxx].

Return:
`unsigned integer` encoded with the message type.

Function:
`uchar *`
`msg_get_field (void)`

Description:
This function returns the next field from a message. The message that it will work on is that which was last accessed by `msg_get_type()`. Each subsequent call to this function will return the next field (either code or value) as a null terminated string, until the end of the message where a NULL pointer value is returned.

Arguments:
none.

Return:
`array of characters` (null terminated) that contains the field.

Function:
`void`
`msg_add_type (uchar * message, uint message_type)`

Description:
This initialises the `message` by inserting the message type onto the front of it (which is defined in [xx] to be of the format ",(type)"). The contents of `message` will be overwritten. Note that no checking is done on the length of `message`.

Arguments:
`message` is an array of characters in which the message type will be placed. `message_type` specifies the type of message.

Return:
none.

Function:
`void`
`msg_add_field_char (uchar * message, uchar * code, uchar value)`

Description:

Appends a new field to `message` with a particular field code and a `value` that is represented as an ASCII character. Note that no checking is done on the length of `message`. This field will actually be encoded as ",(code),(value)", e.g. ",1,Q".

Arguments:

`message` is an array of characters which will have the field appended. `code` is a null terminated array of characters containing the field code. `value` is an ASCII character to be encoded.

Return:

none.

Function:

```
void
msg_add_field_int (uchar * message, uchar * code, uint value)
```

Description:

Appends a new field to `message` with a particular field code and a `value` that is represented as an unsigned integer. Note that no checking is done on the length of `message`. This field will actually be encoded as ",(code),(value)", e.g. ",A,54".

Arguments:

`message` is an array of characters which will have the field appended. `code` is a null terminated array of characters containing the field code. `value` is an unsigned integer value to be encoded.

Return:

none.

Function:

```
void
msg_add_field_str (uchar * message, uchar * code, uchar * value)
```

Description:

Appends a new field to `message` with a particular field code and a `value` that is represented as a null terminated array of ASCII characters. Note that no checking is done on the length of `message`. This field will actually be encoded as ",(code), (value)", e.g. ",A,display string". There must not be any commas in `value`.

Arguments:

`message` is an array of characters which will have the field appended. `code` is a null terminated array of characters containing the field code. `value` is a null terminated array of characters to be encoded.

Return:

none.

Function:

```
void
msg_add_field_real (uchar * message, uchar * code, double value)
```

Description:

Appends a new field to `message` with a particular field code and a `value` that is represented as a real number 'double'. Note that no checking is done on the length of `message`. This field will actually be encoded as ",(code), (value)", e.g. ",A,123.1234567". You may also call this function using a C style 'float', a conversion will be applied.[1]

Arguments:

`message` is an array of characters which will have the field appended. `code` is a null terminated array of characters containing the field code. `value` is a double to be encoded.

Return:

none.

Example:

The following will encode a status indication message:

```
typedef enum
  {
    s_uninitialised = 0,
    s_initialising = 1,
    s_ready = 2,
    s_starting = 3,
    s_online = 4,
    s_stopping = 5,
    s_shuttingdown = 6,
    s_halted = 7
  } state_type;

state_type  current_subsystem_state;

...

bool
send_status_indication (void)
  {
    uchar msg[FRAME_MSG_SZ];

    msg_add_type (msg, MSG_STATUS_IND);
    msg_add_field_char (msg, "A", current_subsystem_state + '0');
    if (current_subsystem_state == s_halted)
        msg_add_field_char (msg, "B", halted_reason + 'A');

    return (frame_send (comms_ctx, msg));
  }

...
void
process_messages (uchar * msg)
  {
```

[1] I'm fairly sure that ANSI-C specifications state that when passing parameters to other functions, all floats are promoted to doubles and interpreted as doubles irrespective of the called functions argument declarations.

```
    switch (msg_get_type (msg))
      {
        case MSG_EMERGENCYSTOP_REQ:
            ...
            halted_reason = h_estop_req;
            change_to_new_state (s_halted);
            ...
...
```