IOCOM library

Connecting IoT/IO devices to control computer over TCP/IP or serial port.

11.9.2019/Pekka Lehtikoski

Table of Contents

1. Introduction

190527, updated 1.9.2019/pekka

The IOCOM library implements communication between a control computer and IO devices over TCP network or full duplex serial communication wire. The library supports scalable range of functionality. Minimum configuration supports only serial communication on plain metal, while some applications may require multi-threading and secured communication over Internet.

Micro-controllers can be used as plain metal, or equipped with Arduino derived libraries, FreeRTOS (currently testing), possibly RT-Thread (under consideration), LWIP for networking, and TLS library. The IOCOM is not bound tightly to any underlying system, so it can be ported basically to any environment with sufficient resources for the task. Still, using configuration which has been implemented and tested makes things much easier.

The same code runs on Windows and Linux. Linux on should cover Raspberry PI and similar devices. We are considering MAC and Android testing as well.

Communication security is becoming paramount in IoT and Manufacturing environments. Thus TLS integration, strong encryption and device identification are essential part of the project.

1.1 Design goals and miscellaneous ideas

180812, updated 1.9.2019/pekka

The IOCOM library connects IO devices to control computer in a heterogeneous network environment. The same library code run on computers and micro controllers.

- Connect distributed IO devices to controller
- Make it simple, reliable and fast.
- Open source, free to use also as part of commercial code.
- No compromises. Only one simple way to the specific job.
- Reference implementation and documented protocol to connect the devices.
- Code can be ported to any system with standard C compiler and sufficient resources.
- Data is seen as memory blocks. Memory blocks are plain byte arrays.
- Memory blocks are transferred to one direction only.
- Application never locks up. If communication doesn't keep up with changes by application, frames are dropped.
- Preserve order of changes and data transfer synchronization, so that application will know that whole memory block is up to date.
- Low small transaction latency. Typical goal is around 1 milliseconds, depending on hardware.
- High through output in bigger transactions.
- Large address space size. Typically 32k bytes for 8/16 bit micro-controllers and 2GB for 32/64 bit systems. Practical maximum memory block size is set by available system memory.
- Point to point connection over TCP socket or full duplex serial communication.
- Low overhead. Small message headers, transfer only changes, compress transferred data.
- Secure TLS connections and device identification when needed.
- RTOS multi-threading support, when needed.

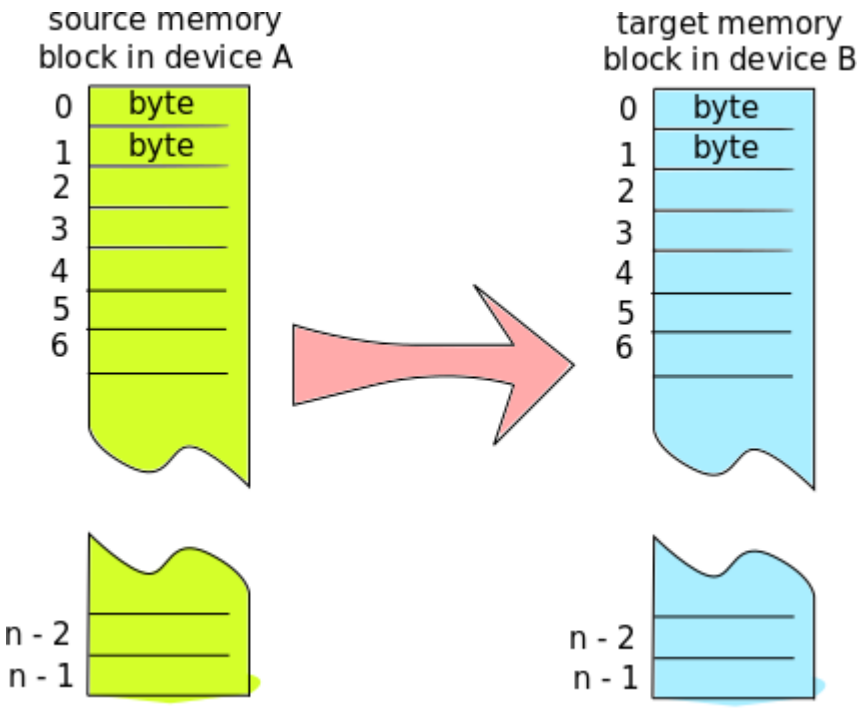
1.2 Memory blocks

180727, updated 22.6.2019/pekka

Communication is based on memory blocks. Device A stores data to be transferred into memory block.

Communication maintains identical copy of the memory block in device B. Separate memory blocks are necessary to implement two directional communication.

A memory block is byte array which is copied from a device to another. Under the hood his is quite a bit more complex, since only changes are transferred over communication, there is compression and synchronization, but user of this library should not need to care about this. There can be multiple recipients of one memory block. There are iocom API functions for allocating, reading and writing memory blocks, and for data transfer synchronization.



A memory block in device A is copied to identical memory block in device B. If the memory blocks have different size, the extra data is ignored.

Communication between two devices typically involves several memory blocks, for example IO board blocks could be "binary inputs", "analog inputs" and "binary outputs". When deciding how to divide data into memory blocks, consider also synchronization and how often input data changes. If some data changes quickly and some is static, these should be in different groups. Data which need to be synchronized (transferred in specific order), like "my data array" and "data array ready" bit, need to be in the same memory block.

A memory block is presented by iocMemoryBlock structure. The structure stores pointers to buffers, flags and state information. A memory block structure is initialized or allocated by:

```
iocMemoryBlock *ioc_initialize_memory_block(
    iocMemoryBlock *mblk,
    iocRoot *root,
    iocMemoryBlockParams *prm);
```

The root object is root of iocom instance's object hierarchy. It kind of binds memory blocks, connections and source/target buffers together. The root object must be initialized before it can be used, and releasing root object will free all resources allocated within the root.

If *mblk* argument is OS_NULL, the function allocates memory for the memory block structure. Otherwise it initializes a memory block in application allocated memory.

Once a memory block is no longer needed, it should be released by calling ioc_release_memory_block function. If the memory block was allocated by the ioc_initialize_memory_block function, the allocated memory is released by this function.

```
void ioc_release_memory_block(
    iocMemoryBlock *mblk);
```

1.3 Access memory block content

190622, updated 23.6.2019/pekka

Memory block content is the actual data to be transferred. It is accessed using functions, which change the content and cause changes to be propagated.

1.3.1 Writing data to be transferred

The ioc_write is the generic function to set data to be transferred.

```
void ioc_write(
    iocMemoryBlock *mblk,
    int addr,
    os_uchar *buf,
    int n);
```

There are simple funtions to set one typed value. These set 8 bit integer value (1 bytes), 16 bit integer value (2 bytes), 32 bit integer value (4 bytes), 64 bit integer value (4 bytes) and 32 bit floating point value (4 bytes). These functions take care that values are stored least significant byte first order within memory block, regardless of processor architecture. These functions convert longer (or same length) integer type, so we do not need to care wether value is signed or unsigned.

```
void ioc_set8(
    iocMemoryBlock *mblk,
    int addr,
    os_int value);

void ioc_set16(
    iocMemoryBlock *mblk,
    int addr,
    os_int value);

void ioc_set32(
    iocMemoryBlock *mblk,
    int addr,
    os_int value);

void ioc_set64(
    iocMemoryBlock *mblk,
    int addr,
    os_int64 value);

void ioc_setfloat(
    iocMemoryBlock *mblk,
    int addr,
    os_float value);
```

1.3.2 Writing string

String is written by ioc_write() function:

```
void ioc_setstring(
    iocMemoryBlock *mblk,
    int addr,
    const os_char *str,
    int n);
```

For example:

```
ioc_clear(mblk, addr, “teststrings”, 10);
```

The example above writes always 10 bytes to wipe out any previous value. Resulting string in memory block has maximum nine characters followed by at least one ‘\0’ character. Strings should be UTF-8 encoded and always terminated with ‘\0’ character.

1.3.3 Writing arrays

Writing arrays element at a time is somewhat waste or processor time and if IOC_AUTO_SEND flag is set, generates extra data transfers. Use functions to write typed arrays into memory block:

```
void ioc_setarray16(
    iocMemoryBlock *mblk,
    int addr,
    os_short *arr,
    int n);

void ioc_setarray32(
    iocMemoryBlock *mblk,
    int addr,
    os_int *array,
    int n);

void ioc_setfloatarray(
    iocMemoryBlock *mblk,
    int addr,
    os_float *array,
    int n);
```

Here n is number of array elements, not number of bytes. You may notice that array functions for bytes and for 64 bit integers are missing. For simple byte data this is because the function would be the same as ioc_write(). Reason for missing 64 bit array function is that I cannot imagine where it would be needed (can be added if someone really needs it).

1.3.4 Clearing range in memory block

The ioc_clear() function sets range of bytes starting from address given as argument to zero. This is preferred way to clear memory block data.

```
void ioc_clear(
    iocMemoryBlock *mblk,
    int addr,
    int n);
```

1.3.5 Propagating writes to transfers

Propagating changes is done by ioc_send() function. This function is called automatically if IOC_AUTO_SEND flag was given when the memory block was initialized. Similarly received data is propagated by ioc_receive() function, and flag to call it automatically is IOC_AUTO_RECEIVE.

1.3.6 Reading received data

Similarly to writing to outgoing memory block, we can read incoming memory block. Data can also be read from outgoing memory block, but never write to incoming memory block. Generic read function is as:

```
void ioc_read(
    iocMemoryBlock *mblk,
    int addr,
    os_uchar *buf,
    int n);
```

The read functions for simple values are:

```
os_int ioc_get8(
    iocMemoryBlock *mblk,
    int addr);

os_int ioc_get8u(
    iocMemoryBlock *mblk,
    int addr);

os_int ioc_get16(
    iocMemoryBlock *mblk,
    int addr);

os_int ioc_get16u(
    iocMemoryBlock *mblk,
    int addr);

os_int ioc_get32(
    iocMemoryBlock * mblk,
    int addr);

os_int64 ioc_get64(
    iocMemoryBlock *mblk,
    int addr)

os_float ioc_getfloat(
    iocMemoryBlock *mblk,
    int addr);

void ioc_getstring(
    iocMemoryBlock *mblk,
    int addr,
    const os_char *str,
    int n);
```

When reading, we may convert to longer integer type, thus sign conversion matters. Thus there are unsigned version of ioc_get8u, ioc_get16u and ioc_get32u. 64 bit integers are assument to be always signed.

1.3.7 Reading arrays

Same as for writing, data can be read as arrays. Here n is number of elements, not number of bytes.

```
void ioc_getarray16(
    iocMemoryBlock *mblk,
    int addr,
    const os_short *arr,
    int n);

void ioc_getarray32(
    iocMemoryBlock *mblk,
```

```
    int addr,
    const os_int *array,
    int n);

void ioc_getfloatarray(
    iocMemoryBlock *mblk,
    int addr,
    const os_float *array,
    int n);
```

1.3.8 Detecting received data using callback function

Using callback function to react to received data is usually much more efficient than polling for changes. Application implemened callback function could be something like:

```
static void iocontroller_callback(
    struct iocMemoryBlock *mblk,
    int start_addr,
    int end_addr,
    os_ushort flags,
    void *context)
{
    /* Echo 2 bytes at address 2 back to IO board address 11. This happens
       practically immediately.
     */
    if (end_addr >= 2 && start_addr < 2 + 2)
      {
       os_int command_echo = ioc_get16(mblk, 2);
          ioc_set16(c->outputs, 11, command_echo);
      }
}
```

Use ioc_add_callback to set the callback function:

```
ioc_add_callback(mblk, iocontroller_callback, OS_NULL);
```

There are few things to be aware of when using callbacks:

- Callback must return almost immediately, it cannot have long processing and even debug prints slow down communication significantly. If callback needs to initiate loger process, trigger an event or set a flag from the callback function.
- In multithread operation, the callback function can is called by other thread than which runs the sequence. Typically thread running the communication.

1.3.9 About thread safety

If multithreading support is enabled for eosal and iocom when compiling, memory block access is thread safe. These functions can be called from multiple threads.

1.4 Devices, memory blocks and connections

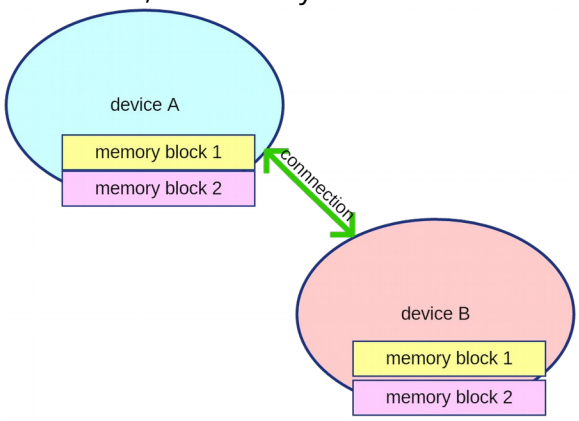
190502, updated 21.6.2019/pekka

A device holds multiple memory blocks, at minimum one for receiving data and another for sending data. There can be also memory blocks for IO device structure information (metadata), for programming flash, etc. Basic communication doesn't really care what memory block is for.

1.4.1 Connecting two devices together

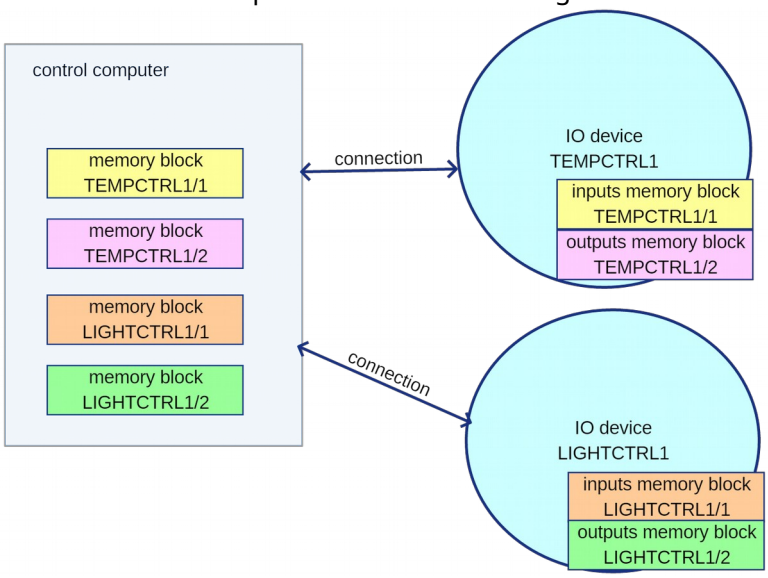
A simple example is connecting device A and device B. Both devices "know" that they have memory blocks 1 and 2,

and these memory blocks are constructed by ioc_initialize_memory_block() calls at both ends. Once connection is established, the memory blocks which have same number will be transferring data.



1.4.2 Connecting multiple IO devices to control computer

Usually we need to add some more complexity. For example we have one control computer, which controls two IO devices: One for temperature and other for lights.



Here the memory blocks are also created before establising the connections between the control PC and the IO devices. Difference is that here in addition to memory block number we give memory blocks also device name and device number. To transfer data between two memory block all three must match.

1.4.3 Device name and number

Device name and device number make an device identifier which is unique within a control system. Device name describes IO board functionality, so two devices with name “LIGHTCTRL” are assumed identical. The device name must be a short string (max 12 characters) with only upper case characters ‘A’ - ‘Z’. Since one control system may have multiple same kind of IO deviced, these are separated by device number. Usually 1 if first device... The device name is typically hard coded in device software and device number (if needed) either set by jumpers or stored in EEPROM.

1.4.4 Who connects and who listens for connections?

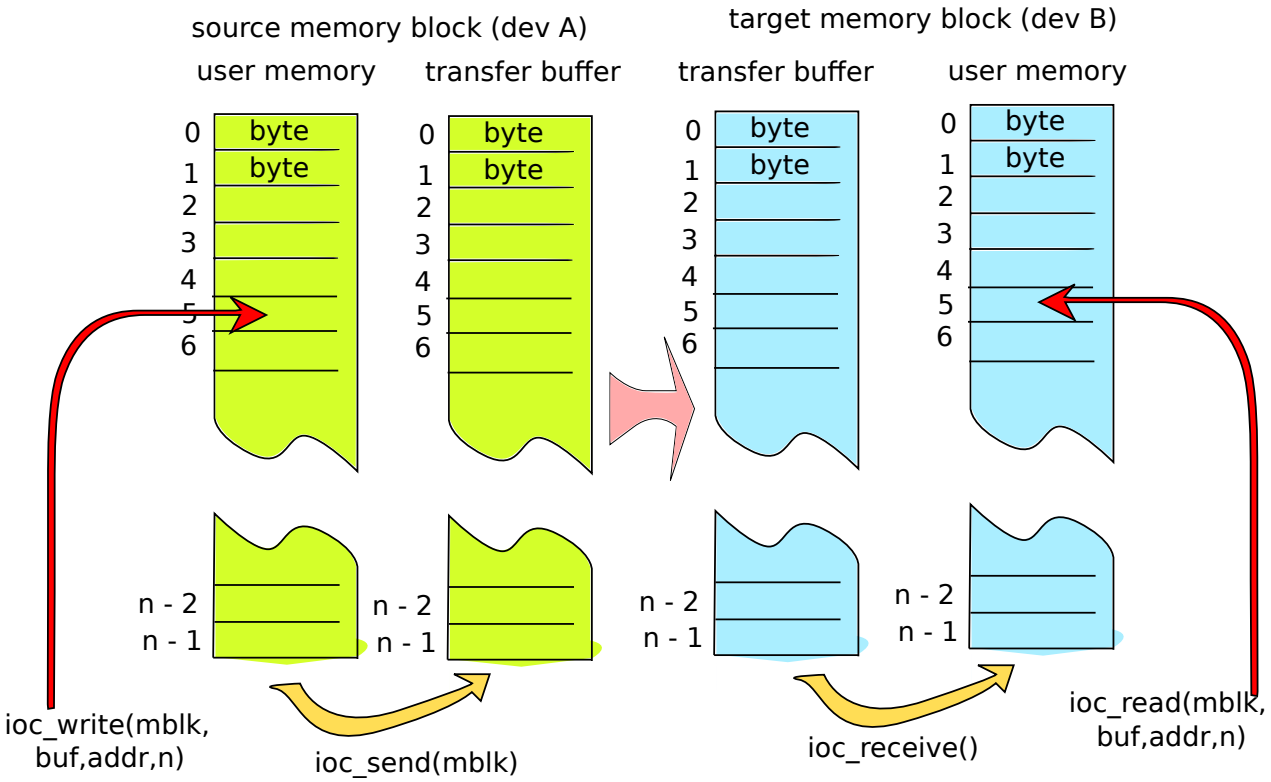
It doesn’t really matter which end listens for socket connection, or if connection is trough serial port. It works just the same. One good choice in own subnet is to assume that IO board “knows” IP address of the control computer and connects to it with TCP socket. The “known” address could be calculated from device’s IP address and subnet mask, for 255.255.255.0 network this could be said “first theree numbers are same as in my own IP address, and

the fourth number is always 253”. If in addition DCHP is used to get IO device’s network configuration, this would work like plug and play without using UDP broadcasts/multicasts. This approach works also well with planned future TLS connection.

1.5 Data synchronization

190502, updated 20.6.2019/pekka

The synchronization logic preserves time order of changes. Consequent changes are received in same order as they were set or in same snapshot. Never in reverse order.



Transfer buffer drawn above here is simplified, implementation detail like real record keeping, buffering and transfer format are not discussed here.

Memory block content is accessed from application code using ioc_write() and ioc_read() functions, or simple ioc_set*() and ioc_get*() functions. Target buffer is read only and source buffer is read/write. On source side, the ioc_send() function copies source memory block to be transferred (or at least modified part of it). If there no space in transfer buffer, an ongoing transfer, the function does nothing. If there are no modifications, the function may do nothing. We can say that ioc_send() records (or may record) a snapshot of user memory for transfer. On target end, the ioc_receive function copies data from transfer buffer to user memory. The copied data is last complete transferred snapshot.

Note about IOC_AUTO_SEND mode: If this flag is given to source memory buffer, ioc_send() is called every time ioc_write() is called. Similarly IOC_AUTO_RECEIVE causes ioc_receive() to be called every time ioc_read() is called. This is not really unefficient, and can be recommended at least to get started.

1.6 MODBUS comparison

190901, updated 1.9.2019/pekka

The library is an alternative to MODBUS protocol for point to point communication. Since the MODBUS is well known, it is easy to get the general idea by listing the main differences between the protocols:

- The IOCOM is event based, while MODBUS uses polling. This means that data is transferred when it changes, not when it is queried.
- Data transfers are synchronized in way that it guarantees that other end of communication receives changes in same order the changes are made.
- IOCOM uses compression, flow control, etc and aims for performance. This makes it more complex than MODBUS. While most MODBUS users write their own protocol implementation, I do not expect this to work well with IOCOM. I recommend using reference implementation, at least as starting point, unless there is need to port the protocol to Python, Java, etc.
- IOCOM requires full duplex serial communication (at least logically), half duplex will not do. It cannot do “one master – multiple slaves” serial communication, every device needs it’s own serial wire. If those features are important, then use MODBUS. It was designed specifically for that scenario and does it well.
- When needed, IOCOM handles secure TLS connections natively. MODBUS is channeled trough SSH to provide secure connection. SSH is typically too heavy for micro controllers, while many more powerful ones can use TLS.

2. Implementing IO device

The IO device is often micro controller based. I think ARM32 architecture is best choice. Development tools are good, programming is easy, chips range from very cheap to powerful. Having only one architecture to support is great, saves time and money, and a lot. Developers can use same code in different products and need only to learn one assembler, set of development tools etc.
My personal favorite is STM32 chips, because STM has put much effort on development boards (Nucleo/Discovery), development tools, examples and ready libraries. Atollic True Studio with Cube is free and good and easy. STMDuino is alternative which keeps code simpler than Atollic. Good choice it if you have Arduino background, just debugging is more cumbersome and using Cube for pin configuration is not as straight forward.

Big choice is weather to write code in single thread mode directly on metal, or to use RTOS, etc, to get multi threading. Both have advantages. Generally if your team of programmers are solid with concurrent programming and know pitfalls of it, RTOS is probably the right choice. On the other hand, if there is lack of knowledge/experience, use of RTOS is likely to result in boards which crash and burn at random. Direct metal is also the choice, if resources in micro controller are limited. IOCOM library can be used in both these configuration, single thread metal is supported and multi-thread support comes naturally since we run same code on PC.

2.1 Using the library in IO board

190421, updated 30.5.2019/pekka
An IO board communication is typical use for the IOCOM library. I usually power IO boards by ARM32 micro controller. Resources are limited and reliability is the key requirement. Typical choices are:
- We do prefer single threaded model over multi threaded. A programming which crashes the micro controller, locks up the network or shifts timing is far more likely to pass through testing unnoticed in multi threaded programming than in single threaded.
- We stay away from heap based memory allocation, since memory leak or fragmentation could cause reliability problems over long run. Rather memory is allocated globally, from stack or from application managed memory pool.

2.1.1 API for an IO board

The ioc_ioboard.c collects IOCOM functionality needed by an IO board into simple to use API calls.

2.1.2 Single thread - IO board software organization

If IO board code runs on bare metal, it is single thread mode: typical initialization and main loop organization is used. Main loop must never pause. The IOCOM can also be used in multi threaded with RTOS, etc. See ? for information.

2.1.3 Mapping the memory blocks to inputs and outputs

Unless there is significan benefit in bit packing, it is easiest to use 1 digital input or output = 1 byte in memory block. Support for bit packing digital inputs and outputs adds some complexity. In most cases, there is too small performance benefit compared to added complexity.

2.1.4 State machines

In single thread mode we can never call sleep() etc. State machines are simple way to generate code which will execute a timed sequence or loop without pausing the IO board’s main loop.

2.2 Communication status

1890529, updated 29.5.2019/pekka

Communication status refers to general communication information and settings. For example number of connections (sockets, etc) connected to a memory block. In future this could indicate which input data is selected in redundant communication, etc. Communication status may include also settings.

From application's view communication status appears the same as data memory and is accessed using the same ioc_read(), ioc_get16(), ioc_write(), ioc_set16(), etc. functions. For data memory, the address is positive or zero, status memory addresses are negative.

address	*nro bytes*	*description*
IOC_NRO_CONNECTED_STREAMS	2	Number of connected streams at this moment.
IOC_CONNECTION_DROP_COUNT	4	How many times socket connection has been dropped.

For example IOC_NRO_CONNECTED_STREAMS is defined as address -2 and has two bytes, thus less significant byte is is in address -2 and more significant byte in address -1.

2.2.1 Example, print communication status without callbacks

The example function below: Every time a socket connects or disconnects to this "IO board", this function prints number of connected sockets and how many times a socket has been dropped (global count).

```
static int
    prev_nro_connections,
    prev_drop_count;

static void ioboard_show_communication_status(void)
{
    int
        nro_connections,
        drop_count;


    char
        nbuf[32];


    nro_connections = ioc_get16(&ioboard_fc, IOC_NRO_CONNECTED_STREAMS);
    drop_count = ioc_get32(&ioboard_fc, IOC_CONNECTION_DROP_COUNT);
    if (nro_connections != prev_nro_connections ||
        drop_count != prev_drop_count)
    {
        osal_console_write("nro connections = ");
        osal_int_to_string(nbuf, sizeof(nbuf), nro_connections);
        osal_console_write(nbuf);
        osal_console_write(", drop count = ");
        osal_int_to_string(nbuf, sizeof(nbuf), drop_count);
        osal_console_write(nbuf);
        osal_console_write("\n");


        prev_nro_connections = nro_connections;
        prev_drop_count = drop_count;
    }
}
```

2.2.2 Example, get communication status from callback

The iocontroller_callback() example function is called when changed data is received from connection or when connection status changes. Here we are interested in case when connections status, number of connected sockets, etc. or connection drop count changes. We check by callback start and end addressed if either of the two has changed.

```
static volatile int
    nro_connections,
    drop_count;

static void iocontroller_callback(
    struct iocMemoryBlock *mblk,
    int start_addr,
    int end_addr,
    os_ushort flags,
    void *context)
{
    if (end_addr >= IOC_NRO_CONNECTED_STREAMS && start_addr < IOC_NRO_CONNECTED_STREAMS + 2)
    {
        nro_connections = ioc_get16(mblk, IOC_NRO_CONNECTED_STREAMS);
    }

    if (end_addr >= IOC_CONNECTION_DROP_COUNT && start_addr < IOC_CONNECTION_DROP_COUNT + 4)
    {
        drop_count = ioc_get32(mblk, IOC_CONNECTION_DROP_COUNT);
    }

}
```

No heavy processing or printing data should be placed in callback. The callback should return quickly, the communication must be able to process all data it receives. Thus here we just copy the status into global variables.

2.3 Publish static IO device information

190708, updated 8.7.2019/pekka

There are scenarios, where hard coding memory block map in both ends is not the best thing. For example if we wish user interface to display the state of any unknown IO board, or to use names for inputs/outputs, etc, instead of addresses within memory block. This map for example state that “HEATER_POWER” is a 4 byte floating point number at address 121 within memory block “OUT”.

Here, we work at low abstraction level, thus we do not yet define what this information is nor how it is formatted. For now it is enough that this is “const static char mydevice_info[] = x.... ”, basically statically allocated block of memory whose contents do not change, or at least changes are not synchronized.
Recommendation is that device information would start with the name of the device information format as ‘\n’ terminated string. This could be used by controller to check what format the device information is.

To set up a static memory block, we allocate space for it statically. For micro-controllers it is beneficial to use “const static” C declaration, so that no RAM copy of the static information is needed, it can be used directly from flash.

2.3.1 Publish device information using ioboard.c API

If the we use ioboard.c API to set up the, we store device information pointer and size in parameter structure before calling ioboard_start_communication:

```
const static os_uchar mydevice_info[]
 = "di-1\n"
   "HEAT_POWER,OUT,121,float\n"
   "TEMPERATURE,IN,18,float\n";

ioboardParams prm;
os_memclear(&prm, sizeof(prm));
...
prm.device_info = mydevice_info;
prm.device_info_sz = sizeof(mydevice_info);
```

Here is some very basic device information as text example. Since plain text takes kind of a lot of space, this is not optimal way to present device information. It is beneficial to compress device information as much as possible, since it is usually transferred every time when a connection is established.

The static memory pool needs to be slightly larger, by sizeof(iocMemoryBlock) + IOBOARD_MAX_CONNECTIONS * sizeof(iocSourceBuffer) bytes. We need space for memory block structure and source buffer structure for each connection. We do not need to allocate memory for memory block data nor for source buffer content, since this will be IOC_STATIC memory block.

```
static os_uchar
    ioboard_pool[IOBOARD_POOL_SIZE(IOBOARD_CTRL_CON,
        IOBOARD_MAX_CONNECTIONS,
        IOBOARD_TC_BLOCK_SZ, IOBOARD_FC_BLOCK_SZ)
        + sizeof(iocMemoryBlock)
        + IOBOARD_MAX_CONNECTIONS * sizeof(iocSourceBuffer)];
```

2.3.2 Or call basic IOCOM functions to set up

When a device information memory block is allocated, it needs pointer to and size of static data. Flag IOC_STATIC will tell that the buffer is static and source synchronization buffering is not needed.

```
os_memclear(&blockprm, sizeof(blockprm));
blockprm.mblk_nr = IOC_DEVICE_INFO_MBLK;
blockprm.mblk_name = "INFO";
blockprm.buf = (os_uchar*)prm->device_info;
blockprm.nbytes = prm->device_info_sz;
blockprm.flags = IOC_SOURCE|IOC_STATIC;
ioc_initialize_memory_block(OS_NULL, &ioboard_communication, &blockprm);
```

2.3.3 Dynamic access to device information in control computer

This is code in control computer's end. A "root callback" function can be set at startup. It gets called when a new memory block is created (among other things).

```
osal_socket_initialize();
ioc_initialize_root(&root);
ioc_set_root_callback(&root, root_callback, OS_NULL);
```

Creating dynamic memory blocks is be enabled by IOC_DYNAMIC_MBLKS flag in either ioc_listen() or ioc_connect() call.

```
ep = ioc_initialize_end_point(OS_NULL, &root);
os_memclear(&epprm, sizeof(epprm));
epprm.flags = IOC_SOCKET|IOC_CREATE_THREAD|IOC_DYNAMIC_MBLKS;
ioc_listen(ep, &epprm);
```

Thus application's root_callback() function gets called with IOC_NEW_DYNAMIC_MBLK event, once connection is establised and memory block information is received from the IO device.

```
static void root_callback(
    struct iocRoot *root,
    struct iocConnection *con,
    struct iocMemoryBlock *mblk,
    iocRootCallbackEvent event,
    void *context)
{
    os_char text[128], mblk_name[IOC_NAME_SZ];

    switch (event)
    {
        case IOC_NEW_DYNAMIC_MBLK:
            ioc_get_memory_block_param(mblk, IOC_MBLK_NAME,
                      mblk_name, sizeof(mblk_name));

            os_strncpy(text, "Memory block ", sizeof(text));
            os_strncat(text, mblk_name, sizeof(text));
            os_strncat(text, " dynamically allocated\n", sizeof(text));
```

```
            osal_console_write(text);

            if (!os_strcmp(mblk_name, "INFO"))
            {
                ioc_add_callback(mblk, info_callback, OS_NULL);
                ioc_set_flag(mblk, IOC_AUTO_RECEIVE, OS_TRUE);
            }
            break;

        default:
            break;
    }
}
```

This function again checks if this is device information memory block “INFO”. If so it adds info_callback for the memory block and enables IOC_AUTO_RECEIVE to read data instead of explicit ioc_receive() call. Info callback will be called when device information data has been transferred. This example just prints the device information (string assumed).

```
static void info_callback(
    struct iocMemoryBlock *mblk,
    int start_addr,
    int end_addr,
    os_ushort flags,
    void *context)
{
    os_char buf[128];

    if (end_addr >= 0)
    {
        ioc_getstring(mblk, 0, buf, sizeof(buf));
        osal_console_write(buf);
        osal_console_write("\n");
    }
}
```

3. Classes

190502, updated 19.5.2019/pekka

The IOCOM library is an object oriented design written in plain C.

- The root object is starting point for communication object hierarchy. If it is deleted, all resources allocated by the library are released. Memory blocks store incoming or outdoing data. The application accesses data by reading and writing these memory blocks.
- A connection object represent connection from this process to another process. A connection object corresponds to a socket, etc.
- Transfer buffers are the bridge between the memory blocks and connection objects. These take care about buffering what should be sent, what is being received and about synchronization.

3.1 Root object

180727, updated 18.5.2019/pekka

An iocRoot object is the root of communication object hierarchy, and must be initalized before using any other iocom functions. It holds first and last object pointers for memory block list, connection list, etc.

If the library is compiled to support multi-threading, the root object holds also the mutex to synchronize access to communication object hierarchy.

An application can have multiple root objects to keep communication. This is useful when using multi-threading in client, to make sure that separate socket connects, etc work independently without delays caused by the other.

The ioc_initialize_root() function initializes the root object. The root object can always be allocated
global variable. In this case pointer to memory to be initialized is given as argument and return value is the same pointer. If dynamic memory allocation is supported, and the root argument is OS_NULL, the root object is allocated by the function.

```
void ioc_initialize_root(
    iocRoot *root);
```

The ioc_release_root() function releases resources allocated for the root object. Memory allocated for the root object is freed, if it was allocated by ioc_initialize_root().

```
void ioc_release_root(
    iocRoot *root);
```

Lock functions are are used to lock object hierarchy for the root so it can be accessed only by one thread at the time. Once the ioc_lock() is called by one threads, other threads are paused when they ioc_lock(), until the first thread calls ioc_unlock().

```
void ioc_lock(
    iocRoot *root);

void ioc_unlock(
    iocRoot *root);
```

If the multi-threading support is not enabled for the build, these lock/unlock functions are replaced by empty macros, which do nothing.

3.2 Memory blocks

190502, updated 19.5.2019/pekka

A memory block is byte array which is copied from a device to another.

A memory block in device A is copied to identical memory block in device B. If the memory blocks have different size, the extra data is ignored.

Communication between two devices typically involves several memory blocks, for example IO board blocks could be "binary inputs", "analog inputs" and "binary outputs". When deciding how to divide data into memory blocks, consider also synchronization and how often input data changes. If some data changes quickly and some is static, these should be in different groups. Data which need to be synchronized (transferred in specific order), like "my data array" and "data array ready" bit, need to be in the same memory block.

A memory block is presented by iocMemoryBlock structure. The structure stores pointers to buffers, flags and state information. A memory block structure is initialized by:

```
iocMemoryBlock *ioc_initialize_memory_block(
    iocMemoryBlock *mblk,
    iocRoot *root,
    int mblk_nr,
    os_uchar *buf,
    int nbytes,
    int flags);
```

If *mblk* argument is OS_NULL, the function allocates memory for the memory block structure. Otherwise it initializes a memory block in application allocated memory block structure. Memory block data size is *nbytes.* Data buffer *buf* is optional and can be OS_NULL (buffer allocated by this function). If buf pointer is given, it must be pointer to nbytes of memory. Memory block identifier number mblk_nr identifies which memory block this is, for example IOC_INPUT_MBLK. If application gives either mblk or buf as argument, these must stay in memory until ioc_release_memory_block is called.

Once a memory block is no longer needed, it should be released by calling ioc_release_memory_block function. If the memory block and/or the buffer were allocated by ioc_initialize_memory_block function, the allocated memory is released by this function.

```
void ioc_release_memory_block(
    iocMemoryBlock *mblk);
```

3.3 Connection objects

180727, updated 18.5.2019/pekka

A connection object represents logical connection between two devices. Both ends of communication have a connection object dedicated for that link, serialized data is transferred from a connection object to another.

```
ioc_initialize_connection(iocConnection *con, iocRoot *root);
ioc_release_connection(iocConnection *con);
ioc_attach_memory_block(iocConnection *con, iocMemoryBlock *mblk, iocTransferBuffer *tbuf);
ioc_detach_memory_block(iocConnection *con, iocMemoryBlock *mblk);

ioc_connect(iocConnection *con, char *constr, flags)
ioc_listen(iocConnection *con, flags)
ioc_accept(iocConnection *con, flags)
```

ioc_disconnect(iocConnection *con)
ioc_run(iocConnection *con);

3.4 Transfer buffers

180727, updated 18.5.2019/pekka

Transfer buffer binds a memory block and connection object together. It buffers changes to be sent through the connection.

ioc_initialize_source_buffer(iocSourceBuffer *sbuf, con, mblk);
ioc_remove_source_buffer(iocSourceBuffer *sbuf);

4. The IOCOM library implementation

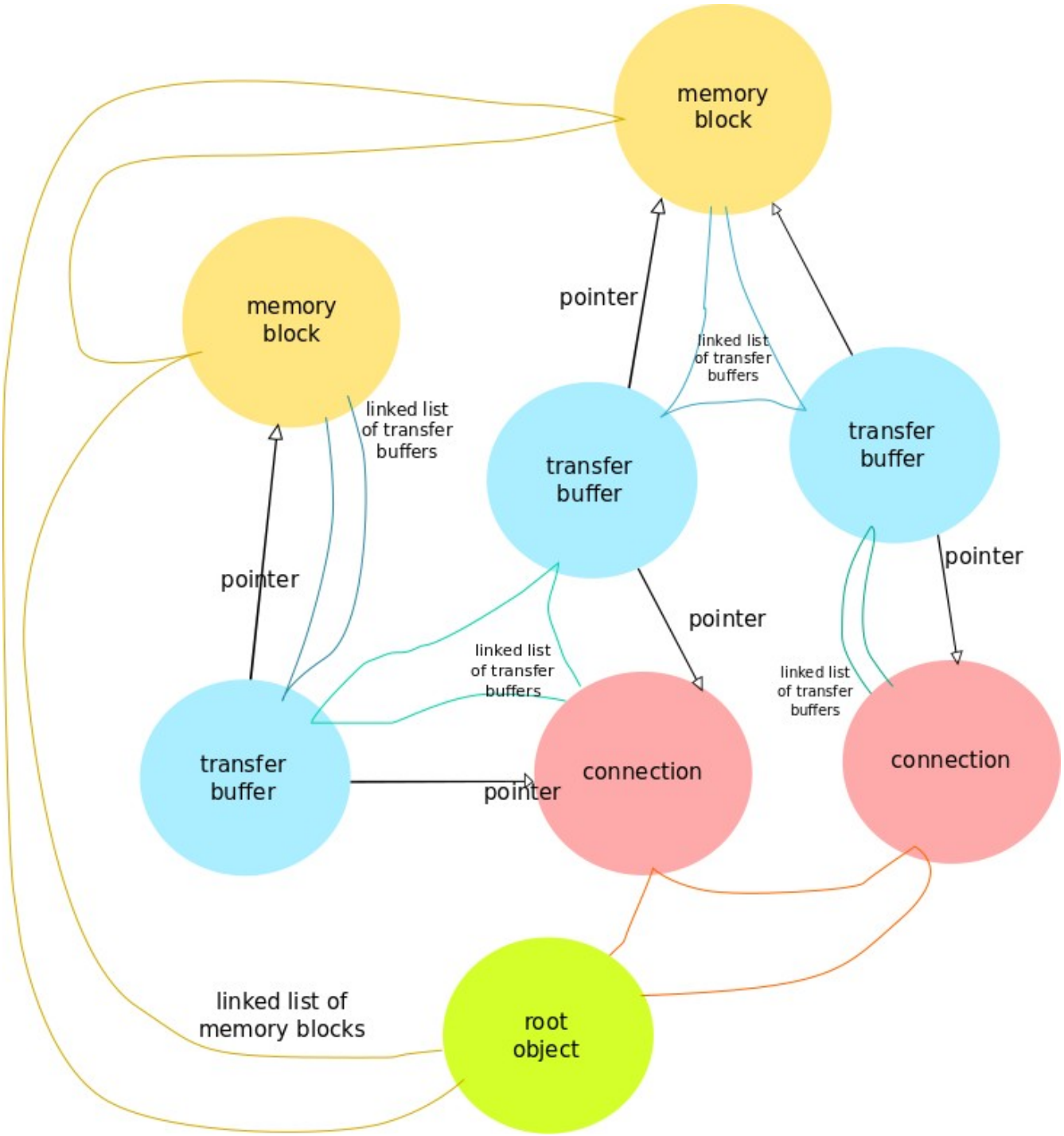
190502, updated 19.5.2019/pekka

The library functionality can be broken into separate concepts. Discussing one topic at a time is more understandable that tying to explain functionality as a single blob. In real code it is these concepts overlap, one .c file rarely corresponds with one topic.

4.1 Object interconnections

180727, updated 18.5.2019/pekka

Linked lists are used to bind communication objects together to form communication hierarchy.

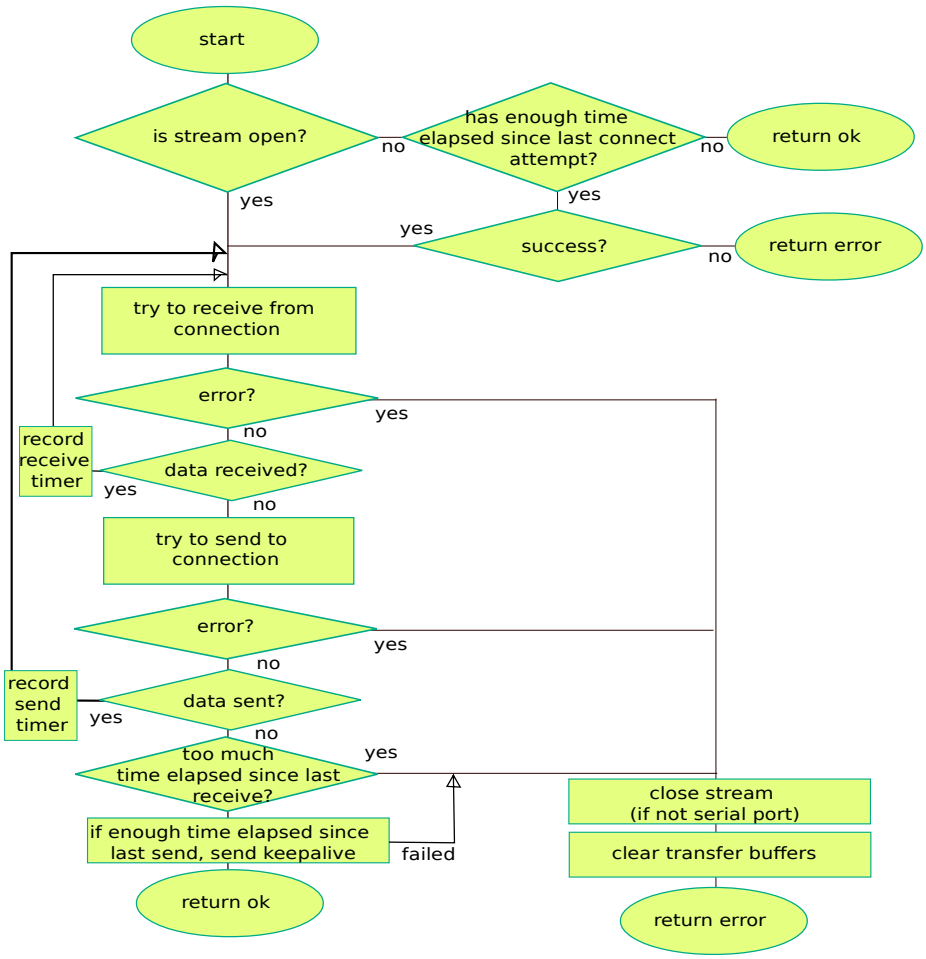


The root maintains list of initialized memory blocks and list of initialized connections. Similarly list of attached transfer buffers is maintained for connections and memory blocks. Object is detached from object hierarchy and all linked lists by calling remove function (like ioc_remove_memory_block()). Transfer buffer structure holds pointers to both memory block and connection it is attached to. In addition (not shown in picture) that every communication object holds pointer to root object.

4.2 Run - keep the connection alive

190502, updated 7.7.2019/pekka

The ioc_connection_run() function keeps the connection the connection alive. This is single thread function which must be called repeatedly. Similar, but more efficient, implementation exists for multi threaded environments mode using select() instead of polling for new data.

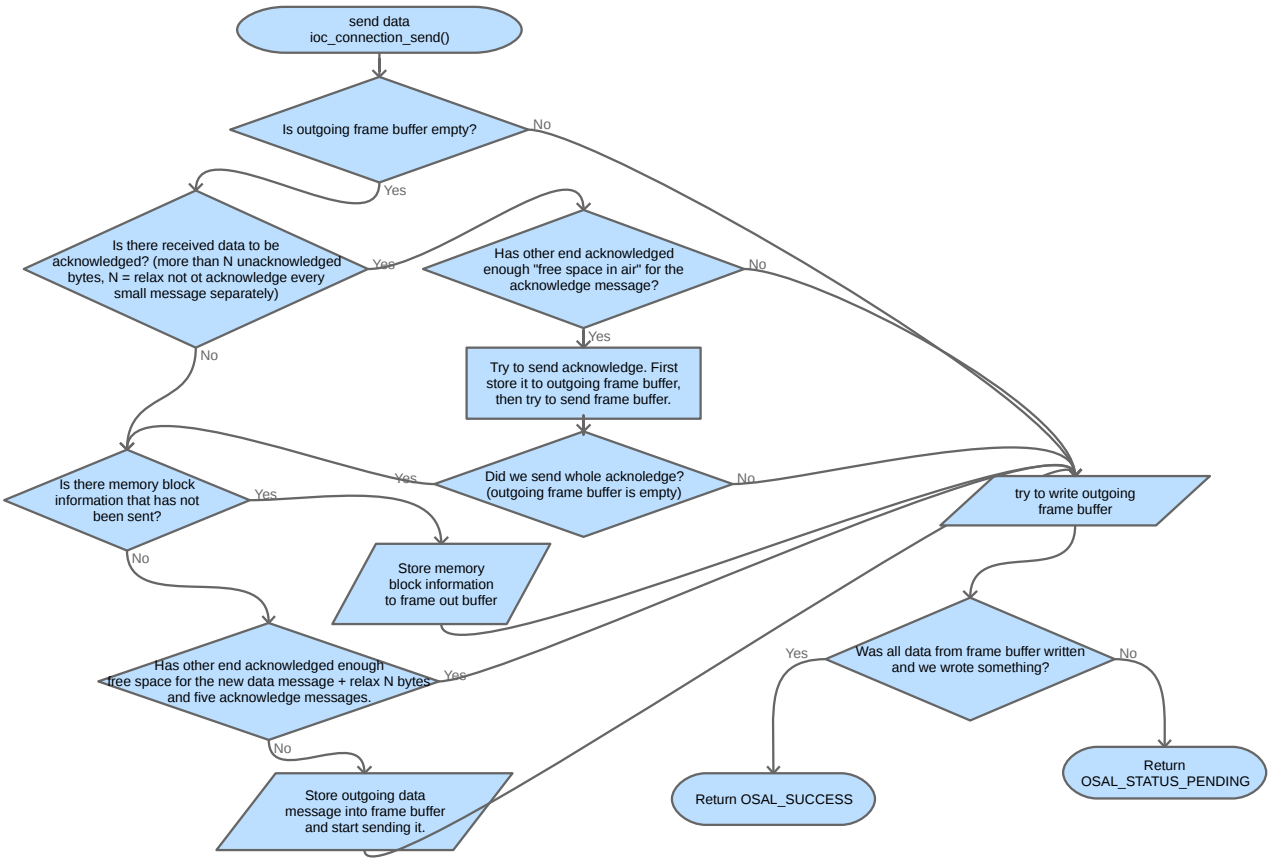


This function calls ioc_connection_send() and ioc_connection() receive functions to send/receive data.

4.3 Sending data

190502, updated 7.7.2019/pekka

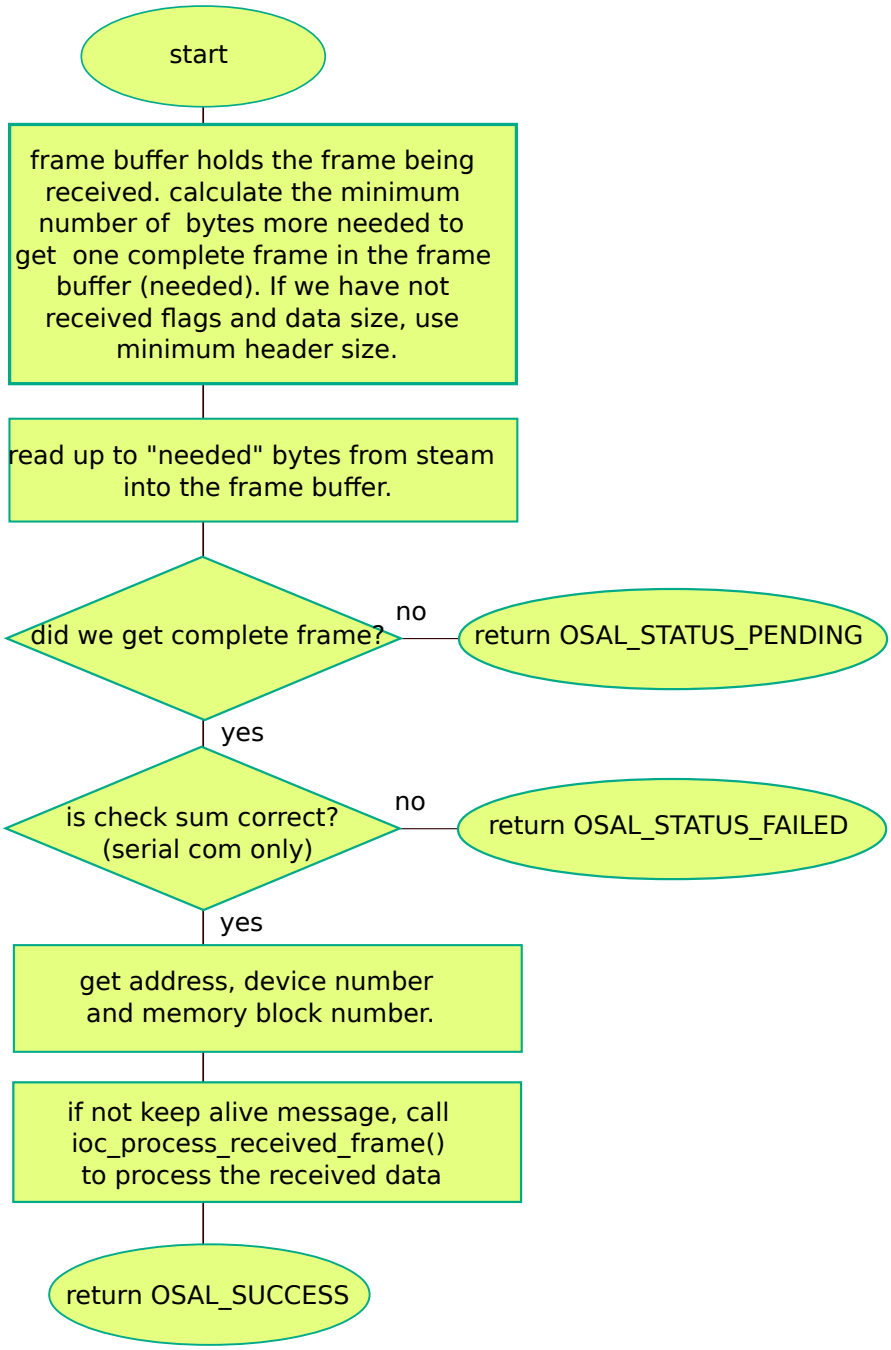
The ioc_connection_send() function flow chart: The function takes care about flow control, sending memory block information and memory block data. Memory block data is moved from source synchronization buffer to frame out buffer, this step includes division of data into frames, compression, etc.



4.4 Receiving data

190502, updated 2.5.2019/pekka

The ioc_connection_receive() function flow chart.



4.5 More about flow control

180910, updated 10.9.2018/pekka

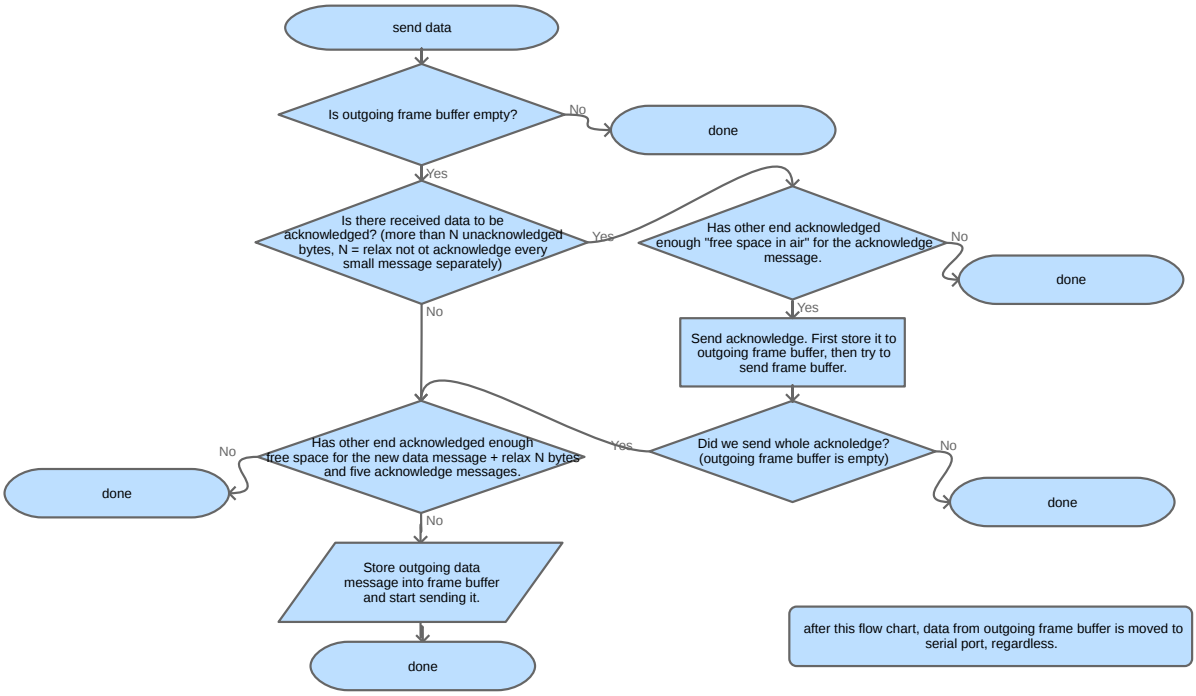
The flow control is used to stop transmitting before serial communication buffers overflow or socket communication buffers collect so much data that communication starts to lag.

Basic idea is simple. A end of connection calculates number of bytes it has sent. The other end calculates number

of bytes it has received and sends this number back as an acknowledgement message (= keep alive message, the two are the same). So sender can know how many bytes it has sent out, but may not have yet been been processed by the other end: Subtract last received RBYTES from number of bytes sent out.

This is done using 16 bit unsigned integers. Since we subtract these from each others, we simply ignore overflow. This works as long as maximum allowed lag is below 65536 bytes. We call the maximum lag parameter MAX_IN_AIR.

Serial receive buffer (256 bytes, 255 effective) is able to hold at least two data frames of IOC_SERIAL_FRAME_SZ (96) bytes, plus we allow IOC_SERIAL_UNACKNOGLEDGED_LIMIT (40) bytes for unacknowledged data, and reserve some space of the buffer to be used only for acknowledgements. Since our minimum acknowledgement is 40 bytes, we acknowledge minimum 200 bytes with five acknowledgement messages, which is more than enough to keep communication up at any load conditions.



Buffers for socket communication are typically large compared to serial communication, and bigger MAX_IN_AIR values can be used. Since sockets do never lose data and can, especially over Internet, have significant latency, large values would be beneficial. On the other hand embedded systems may have limited socket buffers. Thus conservative MAX_IN_AIR value 2000 bytes is for now used for sockets, but may be changed in further testing.

Preventing possibility of dead lock: Common problem with acknowledge based flow control is that if both ends simultaneously have sent out the maximum number of bytes “in air”, but not yet received reply that these have been processed, so both ends halt sending at same time. This has been resolved by reserving space for 5 acknowledgement messages and setting minimum acknowledgement size as 40 bytes.

4.6 Memory block information

190624, updated 24.6.2019/pekka

Each memory block must inform all connected devices about it’s existance. This is needed so that memory blocks in different devices can be bound, resized or dynamically created. Following information is maintained for a memory block.

device name	Device number	Memory block number	Memory block id	Memory bloc name	Memory block size/bytes	Flags (includes direction)
TEMPCTRL	1	1	1	INPUTS	128	0

This information is static type. Let's say, even if memory block is automatically resized, the information here reflects original set values.

When connection is established, memory block information is sent for every existing memory block is sent to new connection. If new memory is added, block information is sent trough every connection.

I connection contains pointer to memory block, whose information is to be sent. When new connection is established, this pointer is set to first memory block. When information about the pointed memory block has been sent trough the connection, pointer is moved to next memory block. When all are ready, pointer becomes zero.

When new memory block if added, we set the pointer within connection object to point it (unless not null, which means that the connection is already working on transferring memory block data).

Thus control computer end is responsible of subscribing data it wants, and informing what data it will send. This generates source and target buffer objects also in device end.

4.7 Data compression

180728, updated 18.5.2019/pekka

Over lossless connection we have to pass only initial state and later on just changes. To compress the only changes we keep record of previous data sent in transfer buffer.

delta[i] = data[i] – prev_data[i]

This is done just with 8 bit integers, overflow ignored. Typically most of diff[i] values are zeroes, since in most cases much of data does not change. So we use zero-run compression:

1. Repeat while we have data left. Interrupt with -1 if compressed data is as long or longer than original data.
2. Count how many nonzero bytes there are starting from current byte p. Count up to maximum 255 bytes. Do not interrupt count to one stand alone zero
3. Write number of nonzero bytes.
4. Write nonzero bytes.
5. Calculate number of zero by
6. Count how many zero bytes there are starting from current byte p. Count up to maximum 255 bytes.
7. Write number of zero bytes.
8. Go back to step two.

p = pointer to data to compress.
bytes = number of bytes to compress.
dst = destination buffer for compressed data.
dst_sz = destination buffer size in byte.
returns number of bytes in destination buffer or -1 if is not compressed (longer than original).

```
int ioc_compress(
    os_uchar *srcbuf,
    int *start_addr,
    int end_addr,
    os_uchar *dst,
    int dst_sz)
{
    os_uchar
        *dst_start,
        *dst_end,
        *p,
        *start;


    int
        bytes,
        count,
        dst_count,
        max_count;


    dst_start = dst;
    dst_end = dst + dst_sz;


    p = &srcbuf[*start_addr];
    bytes = end_addr - *start_addr + 1;


    while (bytes > 0)
    {
        start = p;
        count = 0;
        max_count = bytes;
        if (max_count > 255) max_count = 255;
        while ((p[0] || p[count < max_count-1 ? 1 : 0]) && count < max_count)
        {
            count++;
            p++;
        }
        bytes -= count;


        if (dst + count >= dst_end)
        {
            p = start;
            break;
        }


        *(dst++) = count;
        while (count--) *(dst++) = *(start++);
        if (bytes == 0) break;


        start = p;
        count = 0;
        max_count = bytes;
        if (max_count > 255) max_count = 255;
        while (*p== 0 && count < max_count)
        {
            count++;
            p++;
        }
        bytes -= count;


        if (dst >= dst_end)
        {
            p = start;
            break;
        }
        *(dst++) = count;
    }


    count = p - &srcbuf[*start_addr];
    *start_addr = p - srcbuf;
    dst_count = dst - dst_start;


    return dst_count < count ? dst_count : -1;
```

}

The outgoing message will contain “delta encoding” and “zero run compression” bits in header. If compression result would be as long or longer than data in uncompressed format, compression is not done. The delta encoding is not done at least for the first snapshot to send. Data received should use these bits to detect if data is delta encoded or compressed to read it.

4.8 Uncompress data

This is simpler to implement than compression:

1. Start from first source byte to process.
2. Quit if we have processed all source data.
3. Get next source byte in *n*. It is count of real data bytes.
4. Move *n* data bytes from source to destination.
5. Quit if we have processed all source data.
6. Get next source byte in *n*. It is count of zero bytes.
7. Write **n** zero bytes to destination
8. Back to step 2.

In addition we must check that we do not overflow destination buffer, in case our source data is erroneous. Undoing delta encoding is also simple. We add received delta values to previous values.

```
data[i] = data[i] + delta[i]
```

This is again done just with 8 bit integers, overflow ignored.

4.9 Data transfer framing

180728, updated 22.6.2019/pekka

Serial frame.

name/address	bytes	type	use
FRAME_NR/0	1	uchar	Frame count. Every time new frame is sent, this number is incremented, value 0 if used to indicate the first frame and skipped from that point on. Used to check for missing or duplicated frames, first frame, etc. Value 255 is used as ACKNOWLEDGE/KEEP ALIVE. Also values greater than 220 are reserved for control characters.
CHECKSUM/1	2	ushort	Checksum. 16 bit CRC. Used to verify frame integrity.
FLAGS/3	1	uchar	Flags: IOC_DELTA_ENCODED, IOC_COMPRESESSED, IOC_ADDR_HAS_TWO_BYTES, IOC_ADDR_HAS_FOUR_BYTES, IOC_MBLK_HAS_TWO_BYTES, IOC_SYNC_COMPLETE
BYTES/4	1	uchar	Data size, bytes. Compressed or uncompressed. Data size zero indicates special message. In this case start address is parameter for special message.
MBLK/5	1/2	uchar/ushort	Memory block identifier.
ADDR/6-7	1/2/4	uchar/ushort/uint	Start address within memory block. Or for special messages like keep alive or request to bind, this is 32 bit value containing both special message and argument for it.
DATA/7-11	?	uchar[]	Data. Can be delta encoded and/or compressed.

Socket frame.

name/address	bytes	type	use

FLAGS/0	1	uchar	Flags: IOC_DELTA_ENCODED, IOC_COMPRESESSED, IOC_ADDR_HAS_TWO_BYTES, IOC_ADDR_HAS_FOUR_BYTES, IOC_MBLK_HAS_TWO_BYTES, IOC_SYNC_COMPLETE Value 255 reserved for ACKNOWLEDGE/KEEP ALIVE.
BYTES/1	2	ushort	Data size, compressed or uncompressed, bytes. Data size zero indicates special message. In this case start address is parameter for special message.
MBLK/3	1/2	uchar/ushort	Memory block identifier.
ADDR/4-5	1/2/4	uchar/ushort/uint	Start address within memory block. Or for special messages like keep alive or request to bind, this is 32 bit value containing both special message and argument for it.
DATA/5-9	?	uchar[]	Data. Can be delta encoded and/or compressed.

All numeric data is transferred in small endian form, lest significant byte first and most significant byte last.

Maximum frame sizes for different types of communication are fixed. For all serial communication maximum message frame size is always 96 bytes (IOC_SERIAL_FRAME_SZ), which includes headers. For sockets maximum message frame size is set 464 bytes (IOC_SOCKET_FRAME_SZ). Maximum frame sizes (especially for serial communication) are small to work with limited memory in micro-controllers. Notice that multiple message frames can be packed in single TCP frame.

4.10 Serial communication

180812 – updated 14.6.2019/pekka

Serial communication is differs from socket. Integrity of transferred data must be ensured at protocol level, broken connections nee to be handled, etc. There is also a need to detect beginning of connection, etc. Serial communication bandwidth is usually more limited than in network, and latency may be less of an issue than in large scale network. We treat serial communication as “almost reliable”: we pass only initial state and later on just changes. Communicatin errors are detected, and connection is restarted on error.
imilar to socket, communication transmits changes to IO state immediately without waiting for a query, token passing, etc. Flow control is kept to minimum. This approach gives good troughoutput and low latency. But it puts a few requirements on design.

1. Full duplex connection is required.
2. Device should be able to process received data at serial baud rate. Primitive flow control is provide to avoid collapse of communication on overload, bu if we hit this cealing communication may slow down significantly.
3. In sigle thread mode, the iocom_run() functin should be called repeatedly without long pauses. Maximum tolerable pause can be calculated: Assume line speed 38400bps and serial receive buffer size 256 bytes. 38400 bps is about 3840 bytes per second (8 data bits, one stop bit, parity bit). In numbers 1000ms * 256/3840 = 67ms. Thus in this example case maximum acceptable pause between iocom_run() calls is 67ms. To make it safe we add 10% margin and say 60ms. To work reliably also under full load, the function must be called at least 17 times per second.

Processing capacity of an embedded device is hard to calculate, thus it may also be challenging to be sure of iocom_run() call frequency by reading code. But these are easy to test. For example we are testing IO board we just run controller software in linux/windows computer which writes random data to IO device at maximum line rate. If IO board cannot process it or call iocom_run() often enough, the connection will be limiteed by flow control.

4.10.1 Error detection, handling and keep alive messages

Transmission errors are detected in two main ways. First every frame transmitted over serial connection is enumerated. If these are not received in precise order, the connection is dropped on error. Secondly every frame

contains check sum, which must match to rest of frame content. First frame has FRAME_NR zero. From then on frame number is incremented until IOC_MAX_FRAME_NR, after IOC_MAX_FRAME_NR frame number jumps to 1 (number 0 is never repeated). In code the frame number is maintained in “frame_nr” for both outgoing and incoming data.

If error is detected, a the connection swiches back to “initiate connection” mode. This starts either BY CONNECT or DISCONNECT character. Thus the connection is dropped off and reinitialized. All key frames are retransmitted. This is somewhat time consuming and the connection will pause for a period. This way of handling errors is well suited for serial links which do not normally have errors, like adequately designed wired serial connections. It is not well suited when communication breaks are frequent, for blue tooth, etc.

If no other data is sent, a connected connection sends periodic “keep alive” message to keep other end of connection that all is still working. If no keep alive no other data is received for a period, connection is terminated and moves back to “connection initialization state”. So silent line and communication errors are treated exactly the same way.

4.10.2 Check sums

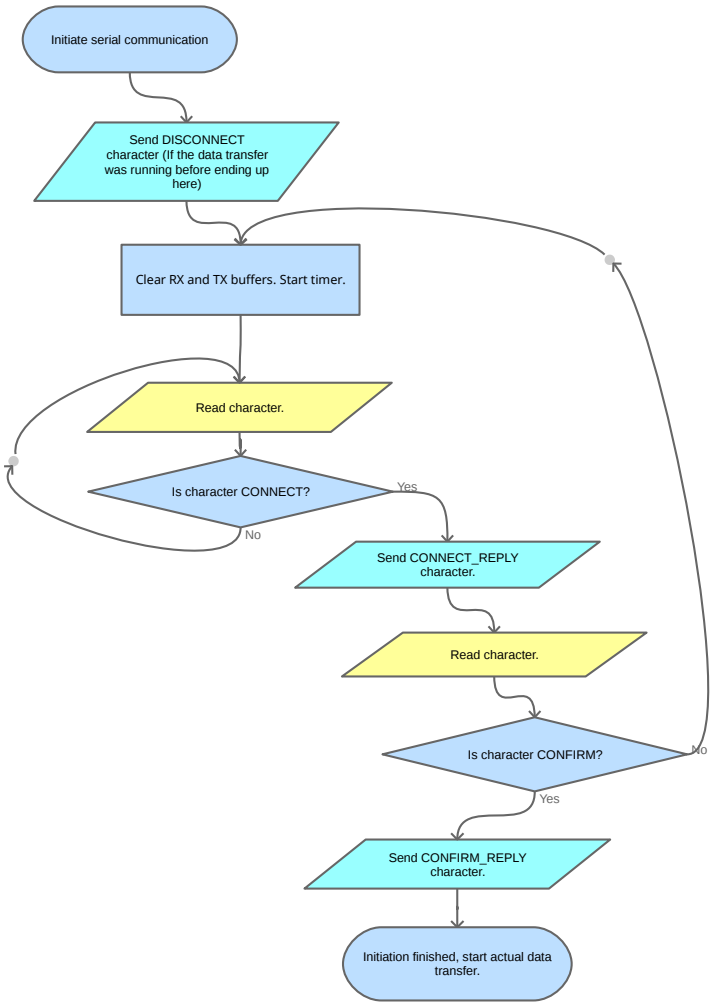
The check sum is calculated from whole transmitted frame, including header. The two bytes used to store the check sum are zeros when the check sum is calculated. Used algorithm is Modbus CRC-16.
See ioc_checksum.c for details.

4.10.3 Establishing connection - client

When client end of connection is in disconnected mode, it periodically sends “CONNECT” character to serial line. Once server end receives the “CONNECT” character, it sends “CONNECT_REPLY” character back... Serial communication connect state machine for client end:

4.10.4 Establishing connection - server

When disconnected, the server end listens for “CONNECT” character from client and initiation goes on from there. Serial communication connect state machine for the server end:



4.10.5 Timing

Timing is defined in milliseconds regardless of connection speed. Reasoning for this is that we use relatively long timeouts which can be easily timed using typical OS time slice. Notice that timeouts effect to communication speed/latency only when there are transmission errors in serial line.

- There may not be long pauses within a message frame. Maximum allowed pause between two characters is 100ms. If there is longer pause than 100 ms, receiver will decide connection is broken and clear receive buffer and restart establishing connection.
- There must be at least 200ms pause between “connect” messages sent by client.

4.10.6 Testing serial communication

I is useful to test new IO device software in PC without actual hardware. Serial communication of a simulated IO device can be tested either in linux or Windows. In Linux one can use socat to create virtual serial ports connected together. For example

- sudo apt-get install socat
- sudo socat PTY,link=/dev/ttyS30 PTY,link=/dev/ttyS31
- cd /dev/pts
- sudo chmod 0666 2
- sudo chmod 0666 3

Would install socat and create and connect ports “ttyS30” and “ttyS31” together. Initially this gives only root access

to the simulated ports. The /dev/ttyS30 and /dev/ttyS31 are links to /dev/pts/2 and /dev/pts/3 (in my test, for you these may be different.

- cd /dev
- ls -la ttyS3*
- output: lrwxrwxrwx 1 root root 10 Jun 3 14:02 ttyS30 -> /dev/pts/2
- output: lrwxrwxrwx 1 root root 10 Jun 3 14:02 ttyS31 -> /dev/pts/3
- cd /dev/pts
- ls -la
- sudo chmod 0666 2
- sudo chmod 0666 3

I have used com0com on Windows. It works nicely, if one gets it set up. But setting it to run on Windows 10 is a hassle. It needs to load unsigned device driver(s).

4.11 Multi-threading and thread safety

180727, updated 18.5.2019/pekka
If multi-threading is enabled for the library, the function calls are reentrant and can be called from different threads.

The define below controls weather multi-threading support is enabled , set either 1 or 0. There are system specific defaults for this, but this can be overridden by setting compiler define.

#define OSAL_MULTITHREAD_SUPPORT 1

4.12 Operating system abstraction layer

180727, updated 7.7.2019/pekka
The IOCOM library needs separate operating system abstraction layer, called eosal library. The eosal wraps operating system and hardware dependent functionality as function call interface. This interface is is similar for all platforms: Purpose of the operating system abstraction is to separate platform dependent code from the bulk of the library. Thus the IOCOM library can run on multiple platforms, like Windows, Linux and several micro-controllers, and be ported to new systems.

4.13 Memory allocation

180727, updated 18.5.2019/pekka
Heap based dynamic memory allocation is used by default in systems that support it. Anyhow using heap based memory allocation is not recommended for micro controllers or other systems with limited memory resources, even if it was supported. I recommend to set up dedicated memory pool in micro controller environments. It can be set up also for linux/windows, which is useful when developing code to run in micro controllers in PC environment.

The OSAL_DYNAMIC_MEMORY_ALLOCATION define (OS abstraction layer, eosal library) is set depending if dynamic memory allocation is supported for the system, either 1 or 0. This usually follows system specific defaults, but this can be overridden by setting the compiler define.

The IOCOM library has it's own functions for memory allocation, ioc_malloc() and ioc_free().
By default these map directly to corresponding eosal library memory allocation functions.

The trick is here: A static memory buffer can be as memory pool for the ioal library.
Function ioc_set_memory_pool() stores buffer pointer within the iocRoot structure.

If pool is set:
- The pool is dedicated to one iocRoot structure.
- All memory needed by the iocom library is allocated from the pool.
- No dynamic memory allocation is used.
- Necessary pool size can be calculated precisely from maximum number of connections, number of registers, etc, communication detail.
- No fragmentation can occur, blocks are never split.

5. Licensing, etc

190502, updated 27.6.2019/pekka

The iocom communication library and operating system abstraction layer, eosal, are published with MIT licensing which allows use in both commercial and open source projects. Basically only requirement this licensing places on you is to include license text with disclaimer in your distribution package (it can be simply like iocom-license.txt readme file).

If you would like to contribute to this project, please see contributor agreement and reasoning for it. If you do not find the contributor agreement acceptable, you can always fork the project.

5.1 MIT license

190625, updated 27.6.2019/pekka

5.2 Contributor agreement explained

190627, updated 27.6.2019/pekka

The contributor agreement gives me and the contributor joint copyright interests in the code. The contributor retains copyrights while also granting those rights to me as the project owner. You need only sign the contributor agreement once in order to cover all changes that you might contribute to any of my projects.

Reason for the contributor agreement is that I might one day use the same code base also for commercial projects, and license those in other terms than MIT license used to publish code as open source. This doesn't mean that MIT licensed code will be taken away or removed. To keep open the option to write also commercial software, I cannot accept any contribution without signed contributor agreement.

Thus I kindly ask to sign to the contributor agreement and email a photo or scan of the completed agreement to pekka.lehtikoski@gmail.com.

5.3 Contributor agreement

190626, updated 27.6.2019/pekka

This Contributor Agreement applies to any contribution that you make to any product or project managed by me, and sets out the intellectual property rights you grant to us in the contributed materials. The term “us” shall mean me, Pekka Lehtikoski. The term “you” shall mean the person or entity identified below. If you agree to be bound by these terms, fill in the information requested below and sign the Contributor Agreement where indicated below.

Read this agreement carefully before signing. These terms and conditions constitute a legal agreement.

1. The term 'contribution' or ‘contributed materials’ means any source code, object code, patch, tool, sample, graphic, specification, manual, documentation, or any other material posted or submitted by you to the project.

2. With respect to any worldwide copyrights, or copyright applications and registrations, in your contribution:
    - you hereby assign to us joint ownership, and to the extent that such assignment is or becomes invalid, ineffective or unenforceable, you hereby grant to us a perpetual, irrevocable, non-exclusive, worldwide, no-charge, royalty-free, unrestricted license to exercise all rights under those copyrights. This includes, at our option, the right to sublicense these same rights to third parties through multiple levels of sublicensees or other licensing arrangements;
    - you agree that each of us can do all things in relation to your contribution as if each of us were the sole owners, and if one of us makes a derivative work of your contribution, the one who makes the derivative work (or has it made) will be the sole owner of that derivative work;
    - you agree that you will not assert any moral rights in your contribution against us, our licensees or transferees;
    - you agree that we may register a copyright in your contribution and exercise all ownership rights associated with it; and
    - you agree that neither of us has any duty to consult with, obtain the consent of, pay or render an accounting to the other for any use or distribution of your contribution.

3. With respect to any patents you own, or that you can license without payment to any third party, you hereby grant to us a perpetual, irrevocable, non-exclusive, worldwide, no-charge, royalty-free license to:
    - make, have made, use, sell, offer to sell, import, and otherwise transfer your contribution in whole or in part, alone or in combination with or included in any product, work or materials arising out of the project to which your contribution was submitted, and
    - at our option, to sublicense these same rights to third parties through multiple levels of sublicensees or other licensing arrangements.

4. Except as set out above, you keep all right, title, and interest in your contribution. The rights that you grant to us under these terms are effective on the date you first submitted a contribution to us, even if your submission took place before the date you sign these terms. Any contribution we make available under any license will also be made available under a suitable FSF (Free Software Foundation) or OSI (Open Source Initiative) approved license.

5. You covenant, represent, warrant and agree that:
    - each contribution that you submit is and shall be an original work of authorship and you can legally grant the rights set out in this contributor agreement;
    - to the best of your knowledge, each contribution will not violate any third party's copyrights, trademarks, patents, or other intellectual property rights; and
    - each contribution shall be in compliance with U.S. export control laws and other applicable export and import laws.

6. You agree to notify us if you become aware of any circumstance which would make any of the

foregoing representations inaccurate in any respect. Pekka Lehtikoski may publicly disclose your participation in the project, including the fact that you have signed the contributor agreement.

7. This contributor agreement is governed by the laws of the State of Georgia and applicable U.S. Federal law. Any choice of law rules will not apply.

8. Please place an “x” on one of the applicable statement below. Please do NOT mark both statements:

   ____ I am signing on behalf of myself as an individual and no other person or entity, including my employer, has or will have rights with respect my contributions.

   ____ I am signing on behalf of my employer or a legal entity and I have the actual authority to contractually bind that entity.

Name*:	
Company's Name (if applicable):	
Title or Role (if applicable):	
Mailing Address*:	
Telephone and Email*:	
Signature*:	
Date*:	
Project Name*:	
Username (if applicable):	

* Required field