librcb4

Open-source library to communicate with the RCB4 board by KONDO from linux

October 2015

By Alfonso Arbona Gimeno (KATOLAB - Nagoya Institute of Technology, Universitàt Politècnica de València)

alf@katolab.nitech.ac.jp, alargi@etsii.upv.es

Supervisor: Professor Shohei KATO

TABLE OF CONTENTS

1.- Introduction....3
2.- Objective and specifications....3
3.- RCB4 communication protocol....4
  3.1.- Physical layer....4
  3.2.- Command protocol....4
4.- Procedure....6
5.- Library implementation....7
  5.1.- Structure....7
  5.2.- Communication....9
  5.3.- Commands....10
    5.3.1.- Common functions....10
    5.3.2.- Source and destination functions....11
    5.3.3.- Special functions and helpers....12
6.- Samples, Makefile and documentation....12
7.- Testing and results....13
8.- Conclusion and usage....13
9.- Release....13
10.- Appendix....14
  10.1.- Appendix A – Full command description....14
  10.2.- Appendix B – Expanded hidden structures....15
11.- References....17

1.- Introduction

The RCB4 is the new core board of the KHR-3 robot by KONDO. The main way of controlling the robot is by using the provided software called HeartToHeart4[i], or using the more advanced library to send commands directly to the robot using its serial interface, however only windows binaries are provided.

Before acquiring this robot researchers at Katolab used the former KHR-2 and thus developed a working library for it that allowed them to create programs in C on linux that communicate with the robot sending commands that set the position of the servos and read the values of the sensors. However, the protocol has changed substantially so it was imperative to create a new library from scratch that allowed porting the old code to the new robot without too much work.

The command reference and robot manual is provided for free in KONDO's website[ii], and using that information a working library was developed using standard linux libraries and system calls that can imitate all the functions provided by those two Windows tools.

It is worth noting that the reference manual is only available in Japanese, and that it contains quite a few mistakes that will, mostly, be documented in the chapter 10.3.- Appendix C – Documentation errata.

2.- Objective and specifications

Although the library for the KHR-2 was intended to be used as a base for the new KHR-3, reading the source code it became apparent that the changes to the protocol are big enough to be forced to develop a new library from scratch. Therefore, instead of trying to create a library with similar calls, new functions will be created trying to make the library easy to use for the developer.

The protocol, explained in the following chapters, is based on a series of commands that get sent to the robot via the serial interface. When the robot receives them sends an answer to the computer, that can be as simple as a simple ACK message, or a whole reply with data (for example if the command asked to read some data from memory). Commands are very variable in contents and length depending on their type and data, so they end up being quite complex.

To help the developer, the library will hide all the data regarding the real command sent to the robot in a private structure that will be filled using different functions. That way, the developer will only have to create the command and call a few functions to fill it with the data to be sent.

On the other hand, connections and serial configuration are also very complex so they will be hidden in another private structure that will also allow different connections to different robots at the same time.

The main objective of the library is therefore to be able to send all the documented commands to the RCB4 board and parse the reply in a way that the developer using the library will find easy to use without the need to understand the communication protocol.

3.- RCB4 communication protocol

3.1.- Physical layer

The RCB4 board uses a simple 3-wire serial interface to communicate with the computer and KONDO provides a USB-to-Serial device to allow easy connection and installation. The device uses the open-source FTDI drivers so it is possible to use it in Microsoft Windows and in Linux.

However there was a problem testing the USB-to-Serial device in my Debian x86_64 (stretch) with libftdi v0.20-3: the kernel did recognize the USB when it got connected but the serial device did not appear. The workaround is to use the older USB-to-Serial for the KHR-2 instead. That device is recognized and can communicate with the robot without problems.

The robot expects all messages to be 8bit+1stop, 5V CMOS **inverted** logic, **even parity** with a command timeout of 150µs. If there is a timeout event the command is silently ignored.

The communication speed is configurable in a special register in the RAM memory (0x0000, bits 6 and 7). Allowed values are 115200bps (default), 625kbps and 1.25Mbps (default in HeartToHeart4's examples). For simplicity only 115200bps and 1.25Mbps are implemented in the library.

To configure the serial the library uses the headers *unistd.h*, *linux/serial.h*, *fcntl.h* and termio.h. It is worth noting that the even parity is mandatory and that the 1.25Mbps is not an standard speed setting so it ended up being necessary to apply a hack to allow that speed[iii].

Furthermore, testing the library it became apparent that there was needed some kind of delay between commands to allow the robot to prepare the reply and answer to our command. The delay applied is 50ms (defined in *rcb4_private.h*) but probably it can be fine-tuned to increase the speed of the library.

The library waits for the answer from the robot in blocking-mode (blocks the execution until a response is received), but has an internal timeout timer of 0.5s (defined in *rcb4_private.h*) to avoid hanging the program if there isn't an answer from the robot (error, disconnection, …).

3.2.- Command protocol

The documentation defines more than 21 different commands available: copy data commands; logical, bitwise and arithmetic operations; execution control commands; servo control orders and utility commands.

All messages start with one byte specifying its length in bytes followed by a byte specifying the type of command. Then there is a variable-length command data that ends in a checksum byte for integrity checks.

This checksum is calculated by adding all bytes in the command until the checksum byte and leaving only the lower byte of the result. For example, the old PING command is 4 bytes, its type byte is 0xFE and there is only one data byte that must always be 0x06, so the checksum is calculated as 0x04 + 0xFE + 0x06 = 0x0108 → 0x08.

The complete list of commands is shown in the following table.

Command	**Length (bytes)**	**Command byte**	**Description**
MOV	Variable	0x00	Copy a value from source to destination.
AND	Variable	0x01	Bitwise AND. Destination = (source & destination).
OR	Variable	0x02	Bitwise OR. Destination = (source | destination).
XOR	Variable	0x03	Bitwise XOR. Destination = (source ^ destination).
NOT	10	0x04	Bitwise NOT. Destination = ~destination.
SHIFT	10	0x05	Left or right shift. Destination = (destination << shifts).
ADD	Variable	0x06	Add. Destination = (source + destination).
SUB	Variable	0x07	Substract. Destination = (source - destination).
MUL	Variable	0x08	Multiply. Destination = (source * destination).
DIV	Variable	0x09	Divide. Destination = (source / destination).
MOD	Variable	0x0A	Modulous. Destination = (source % destination).
JUMP[*]	7	0x0B	Move execution to a ROM address.
CALL[*]	7	0x0C	Call an address in the ROM memory.
RET[*]	3	0x0D	Return from the current function.
ICS	9	0x0E	Set the servos to a set ICS block in RAM.
SINGLE	7	0x0F	Move a single servo.
CONST	Variable	0x10	Move a series of servos all with the same speed.
SERIES[°]	Variable	0x11	Move a series of servos each one with it's own speed.
SPEED / STRETCH[†]	Variable	0x12	Set the speed or stretch of a servo.
Version[*]	3	0xFD	Return the version string from the robot.
PING[*]	3 or 4	0xFE	Check the connection with the robot.

[*] These commands barely change and are very simple so they will be implemented as independent functions that send the message on their own instead of creating a command and sending it like the rest.

[°] Sending this command makes the robot reboot and the HeartToHeart4 program does not send it so it might not be implemented in the robot.

[†] The documentation says the command byte is 0x11 (the same as the SERIES command) and the speed of a servo is already selected in the SINGLE, CONST and SERIES commands so it will not be implemented.

The full protocol description of the commands can be found in the chapter 10.1.- Appendix A – Full command description.

As soon as the RCB4 receives one of these commands it replies with either an ACK/NACK message or with a full reply with data depending on the command issued. The answer starts always with a byte specifying the number of bytes in the reply (including the checksum), the ID of the command that issued the reply, the variable-length data and lastly the checksum of the message calculated as it was for the command.

If the reply is just an ACK/NACK message (does not contain data):

Byte 1 (length)	**Byte 2 (command)**	**Byte 3 (ACK/NACK)**	**Byte 4 (Checksum)**
0x04	Command byte	ACK: 0x06 NACK: 0x15	Checksum

On the other hand, if the message includes data:

Byte 1 (length)	**Byte 2 (command)**	**Byte 3~n**	**Byte n+1 (Checksum)**
n+1	Command byte	Raw data	Checksum

The length of the raw data depend on what the command asked for. If it was a MOV command with a data size of 4 bytes to copy from RAM to COM, the raw data in the reply will be those 4 bytes from RAM. If the command is a logic, bitwise or arithmetic operation the reply will always have the result of the operation.

4.- Procedure

Instead of creating the library directly the first step was to create a simple program able to send a basic command (MOV) to read from RAM.

Using the previous library made by Kazuyoshi Yamada, Yuki Okuzawa and Minoru Ishida for the KHR-2 as an example, a basic communication program was developed. However it did not work at first because there were a lot of differences in the protocol.

Researching how to use the serial interface on linux and reading the manual pages, a basic simple program was created that did send the command but was not able to do much more.

One of the biggest problems faced was that the baudrate changed every day, and later was discovered that another researcher was using the robot with the HeartToHeart4 program, and some examples he uploaded to the robot used one baudrate and others used another baudrate.

Another big problem difficult to find was that the parity is not the usual no-parity, so until that parameter was changed the robot silently ignored all the messages sent.

In the end, the test program ended up being quite big allowing the programmer full access to the MOV command and read the reply of the robot. However, as it was just a test program, the code was not optimum nor clean enough to build the library on top of it, so instead a new clean project was started.

The whole project has been developed using vim and Kate, and to make the development in vim easier the extension *YouCompleteMe* (YCM)[iv] was installed and a makefile rule was created to generate the ctags file with all the autocompletion information.

5.- Library implementation

5.1.- Structure

There are two main blocks in the library. The first one is dealing with the communication and the second is creating the commands.

The communication has to be able to establish a connection to the robot and to send messages to it. In linux that means opening a file descriptor to the serial block device and configuring the serial port; then switching, writing and reading to and from the device; an lastly, restoring the serial configuration and closing the device.

On the other hand, the command creation block has to be able to create any command in a way that it's easy to handle by the user and by the communication library.

Since there are lots of command types very different in structure, one option was to create a single variable type for each one of them, but that would end up being a problem for the library user and for the communication block because it'd have to deal with all the different types of commands.

Another way of dealing with that problem was to make a parser function that translated each command to an array of bytes that the communication block could directly send to the robot. This option helps the development of the library but not much the user because there would still be necessary to have a single variable type for each kind of command.

This problem could be fixed by, instead of creating variables, making all commands a single function that automatically creates the message and sends it to the communication system. This has a very big disadvantage that is having to create a lot of different functions to do the same task because most commands share some parts.

The solution was to create a single complex variable hidden from the user that used extensively unions and packed structures to make sure that the data in memory is exactly how the communication part expects it.

Option	For the user	For the library
Multiple variables	*Difficult*. Different variables not reusable on different commands.	*Difficult*. Lots of functions to do the same and each command has to be sent in its own special function.
Multiple variables with parser	*Difficult*. Different variables not reusable on different commands.	*Normal*. Lots of functions do the same but there only needs to be a send function.
Functions	*Easy*. The use of some commands is extremely easy (just one function call), but the arguments of most functions are difficult and long.	*Difficult*. Lots of functions to do the same. Internally the library is a lot more complex.
Complex variable	*Easy*. Only one variable type. Functions edit that variable setting the corresponding properties of the message.	*Normal*. Designing the structures is hard but the rest is easy. Most functions have only to be created once but their internal code has to deal with the different types of commands.

The main purpose of a library is to be reused lots of times so it is obvious that the main priority is to make the users' life easier even if that implies having to work more to develop the library, so only the last two options were considered. And between those two the second one was easier to implement and to extend in the future if necessary, so that approach was chosen for the library.

To sum everything up, the main structure of the library is a hidden structure with all the connection information and another with the command data. That data is arranged in memory in the same way that has to be sent, so it is extremely easy to develop the communication code. To create the command the user just has to call the corresponding functions that define the message (for example the memory to read and write, the position of the servo, etc).

Hidden structures can only be used outside the library using pointers (because the compiler doesn't know its size as it doesn't know the structure layout) so it will always be necessary a create and a delete function that malloc's and free's the memory. This also gives us the speed of passing variables by reference, that is faster that by value in big structures.

The expanded hidden structures can be read in 10.2.- Appendix B – Expanded hidden structures, and it is worth noting that the command structure must be marked as packed to make sure that the compiler doesn't add any padding to it as it would completely break the way the messages are sent to the robot.

This is because the commands are serialized by just treating the whole structure as an array of bytes instead of a structure. The way the variables are placed in the structure and their size matches perfectly the message to be sent to the robot. If there were padding bytes in the structure the message sent would also have them and the robot wouldn't understand our command.

5.2.- Communication

The communication block, as described in the last chapter, has three main functions: Create the connection, send messages and deinitialize the connection.

To create the connection first the library allocates the variable in memory and opens a file descriptor to the device block in read/write mode (*O_RDWR*), without allowing the terminal to become the controlling terminal (*O_NOCTTY*) and with synchronization active (*O_SYNC*).

Then it saves the configuration of the serial and setups the serial to be able to go at 1.25Mbps using a hack[iii]. By default linux does not allow that speed so this step is mandatory if we want to use a speed other than 115200bps.

The first speed tested is 1.25Mbps because it's the speed used by default in the HeartToHeart4 examples, so the serial is configured to use even parity (*PARENB*), 8 bits (*CS8*), ignore control lines (*CLOCAL*) because there are none, and enable read (*CREAD*). The library also sets up the terminal to work in blocking mode with a minimum of 1 byte returned from *read()* and flushes any messages that might have been in the buffer.

To test if the baudrate is correct the library tries to send twice a ping command. If the robot answers then the speed is correct, if not it changes to the slower 115200bps and tries again the ping command.

The send command function works by directly writing the bytes of the message plus the checksum to the file descriptor, allowing a bit for the robot to send the answer and waiting for it (up to a timeout value).

Depending on if the reply is expected to be a ACK/NACK message or it has data on it two approaches are taken. The ACK/NACK one just reads the reply bytes and parses the message to make sure that it is an answer for our command and that the reply is an ACK. However, if the reply has data then the full message is read and parsed to extract the data. If the user supplied a reply buffer the data is copied there.

Lastly, the deinitialize function just restores the original configuration of the terminal, removes the speed hack, closes the file descriptor and frees the allocated memory.

5.3.- Commands

5.3.1.- Common functions

Commands share most of the functions, and its the function's work to do the appropriate action depending on the type of command. Therefore the first element to set in the command is it's type byte, and once set it must not be changed because it would corrupt its data.

To make this obvious for the user and error-free, the only way the user can choose the command type is when the command is created.

Usually a program doesn't just send a single command, so instead of forcing the developer to create and free the command every time the library provides a recreate function that clears the command (setting everything to zero) and changing the command type if wanted. The allocated space in the create function is already the maximum size of a command so the recreate function doesn't need to call realloc, thus giving a performance improvement and better memory management in exchange on memory usage, but that's a minor problem in this kind of programs and modern computers.

The create command function basically just allocates the memory and calls the recreate function, which just zero's the memory (there are some commands with zero-padding so this is mandatory), sets the command type and, if the command has a fixed length, sets the byte stating the length of the message.

The delete function only frees the memory, and there is also a function to calculate the checksum that just serializes the command, adding all its bytes and then truncating it to an unsigned byte.

Another internal function worth noting is *rcb4_command_get_response_size*. It returns the length of the expected reply based on the command data. Basically it knows if a command only receives an ACK/NACK reply or if it receives data too. And in the case it receives data, it looks up what is the size of that data.

Regarding the shift command, it is not well documented but testing the robot it seems that the correct protocol is to send the number of bits to shift to the left or 256 minus the number of bits to shift to the right. If 0 is sent nothing is changed, and obviously only values from 1 to 127 can be sent. To make it easier for the user (that doesn't need to know the internal way of sending the data) two functions are provided, one for the left shift and one for the right shift.

There are also three different commands that deal with the setting of the servos: set a single servo, set a series of servos all at the same speed and set a series of servos each one with its own speed. To make it easier for the user a single function is created that updates the servo settings differently depending on the command.

If the command only allows a single servo to be moved, calling several times this function just overwrites the servo and its settings. However, the other two commands allow an indeterminate number of servos to be moved at the same time, so the function becomes a bit more complicated.

The only difference between the *const* and *series* is that the former has only a single speed setting and the later has one speed byte for each servo so the pseudocode will be explained only once. Please refer to the Appendix B for more information regarding how the command is saved, but in short it has a 5byte block of flags selecting what servo is going to be edited, and then there is a list of the settings for each set servo in order.

Because the function can be called several times with different servos not in order, the algorithm to update the list of settings becomes a bit more complex. Basically the algorithm developed does the following:

1. Get the block where the new servo is.
2. Go block by block, bit by bit counting the number of servos that are set before our servo.
    1. If the servo we are updating was already in the list, set an overwrite flag.
    2. Else increment the counter of number of servos set.
3. If the overwrite flag is not set, move all the higher servos one position to the right in the list.
4. Write the settings in that position.
5. Set the corresponding bit in the block.
6. Count the number of servos being changed[1,v] and calculate the command length.

The rest of the commands are easy to understand and well documented in the sourcefiles and the Doxygen documentation.

5.3.2.- Source and destination functions

There is an special case for the functions that set the source or the destination of the command because they are very dependent on the type of command.

The code itself is very similar in all cases, but what fields does update inside the structure is completely different, so each function has to handle each command differently. There is also the problem that there are four sources and four destinations completely different to each other, so each one has to have its own function.

There is not a technical difficulty in implementing those functions but they are very long so they have been written in separated source files for clarity.

1 The number of servos being changed is the number of bits set in the 5byte block so the library uses the built-in popcount function because it's the fastest and easiest way to do it. If the CPU implements an assembly function to count the bits it will use it, if not it does implement its own special fast algorithm to count them. There is a bug in most CPU's that make the call of that function in a loop not as fast as it should so the function is called five times and then the results are added. See the endnotes[v] for more information.

It is worth noting that the function does not take into account the endian of the machine and it will cause trouble if compiled in a non little-endian machine. However all computers in Katolab use little endian so the code to take it into account has not been written, but small ToDo comments have been placed in case someone wants to fix it.

5.3.3.- Special functions and helpers

There are a few extra functions to help the user develop and debug programs using librcb4.

If the developer wants to debug a program in ROM, there are three functions to control the flow of the execution: jump, call and return. They behave like their assembly equivalents changing the IP register to the corresponding address in ROM and pushing or popping to/from the stack the corresponding frame.

To print to stdout the full hexadecimal command to be sent there is a function called *rcb4_command_debug_print*, and to make easier the use of delays shorter than 1s, there is *rcb4_util_usleep* that allows to sleep for the desired amount of microseconds. Note that this function calls itself if it exits before the timeout (for example if a signal is sent).

Although it can be achieved using the MOV command, there is a helper function to get the sensor information automatically. It creates its own MOV command and sends it, then reads the reply and writes it to a supplied variable. This way it is extremely easy to develop a program that just reads sensors values and writes to the servos without really having to understand how everything works inside the robot.

6.- Samples, Makefile and documentation

In order to test the library and to make some sample code that a new user could read to understand how the library works, a few sample projects were created. They only show how to use a few of the functions in the library but they are provably the most representative functions: setting up, controlling servos and reading the sensors inside the robot.

To make it easier to create these samples, the makefile searches all .c files and compiles them linking them automatically to the library.

Regarding the makefile, it has been created with scalability and configuration in mind. It automatically adds all source files in *./src/* to the library and includes all files in *./inc/*. The compiled object files are placed in a different folder for clarity and the library is created in the *./lib/* folder.

The makefile also generates the Doxygen documentation when requested using the comments in the source code. Almost all the source files are documented but the only file that will be interesting for a user is the one regarding the file *rcb4.h* because is the only file that the library user has available, however, to make it easier to understand by future library hackers, the documentation also includes the internal functions and files.

7.- Testing and results

To test the library a testing program was created in the samples directory. The main idea was to create automated tests that using random data compared the result of the different functions with known correct values. However the project ended up being too big for the small advantage that it provided so instead of using that program all functions where tested manually.

Tests with the real robot proved that the command to set a series of servos' position and speed did not work and instead made the robot reboot itself. The main reasons for the system rebooting are: there is a problem in the documentation or implementation, or the function is not implemented in the robot.

The rest of the functions proved to work as expected although there were a lot of mistakes in the documentation (see the Appendix C for a list of errors).

8.- Conclusion and usage

The developed library is able to send most of the available commands to the robot, allowing all the functionality that the programs distributed with the robot have. Using this library another student in the laboratory (Kazuki TERABE) was able to adapt a program developed for the KHR-2 that imitates the movements of a human detected by a Kinect to the new KHR-3 with great success.

On the other hand, the library allows to write to COM and to send jump, call and return functions so it is able to be used to create a flashing and debugging program for the robot instead of a communication program.

Apart from the two failing functions in the protocol, a way to improve the library would be to fully develop a way to easily interact with the ICS part of the robot, or making it easier to change different internal settings without having to know their internal address such as the reference value in the AD converters, or the offset of the servomotors.

Regarding the usage of the library, as with almost all libraries in linux, it is necessary to compile it (the Makefile is configured to compile a static library file). Then develop the program including the file rcb4.h (the rest are internal and not needed in a normal program) and then compiling and linking to the library.

It can be summarized to the following steps:

1. make
2. Create the program. `#include “rcb4.h”`
3. Compile your program and link it to the library (for example: `gcc -L/path/to/librcb4/lib -lrcb4 -o my_program my_source.c`).

Keep in mind that you need permission to be able to use the terminal interface. Usually it can be fixed executing the program as root, or by adding your user to the dialout group (preferred method). To achieve this execute as root: `usermod -aG dialout username`, log out and log in.

9.- Release

As this is an academic project that some other institutions can benefit from, the library will be released for free using the open-source license GPLv3 without any guaranty at all except that it was fully created by Alfonso Arbona Gimeno in Nagoya Institute of Technology assisted by professor Shohei KATO.

It will be released using the university's usual way of publishing and in github.com under the username nake90.

10.- Appendix

10.1.- Appendix A – Full command description

byte:	*1*	*2*	3	4	5	6	7	8	9	10~n	n+1
MOV	N+1	0x00	Type		Destination			Source			Checksum
	10						LOW	HIGH	Size	-	
	10						Offset	ICS	Size	-	
	M+7						Byte 1	Byte 2	…	Byte m	
	11						LOW	MID	HIGH	Size	
				LOW	HIGH	0x00					
				Offset	ICS	0x00					
				0x00	0x00	0x00					
				LOW	MID	HIGH					
AND	N+1	0x01	Type		Destination			Source			Checksum
OR	N+1	0x02	Type		Destination			Source			Checksum
XOR	N+1	0x03	Type		Destination			Source			Checksum
NOT	10	0x04	Type		Destination		0x00	0x00	Size	-	Checksum
SHIFT	10	0x05	Type		Destination		0x00	Shifts	Size	-	Checksum
+ (DST = DST + SRC)	N+1	0x06	Type		Destination			Source			Checksum
- (DST = DST - SRC)	N+1	0x07	Type		Destination			Source			Checksum
*** (DST = DST * SRC)**	N+1	0x08	Type		Destination			Source			Checksum
/ (DST = DST / SRC)	N+1	0x09	Type		Destination			Source			Checksum
% (DST = DST % SRC)	N+1	0x0A	Type		Destination			Source			Checksum
JMP	7	0x0B	ROM_L	ROM_M	ROM_H	Conditions	-	-	-	-	Checksum
CALL	7	0x0C	ROM_L	ROM_M	ROM_H	Conditions	-	-	-	-	Checksum
RET	3	0x0D	-	-	-	-	-	-	-	-	Checksum
ICS	9	0x0E	ICS	DataSize	FROM_L	FROM_H	TO_L	TO_H	-	-	Checksum
RunSingleServo	7	0x0F	ICS	Speed	Pos_L	Pos_H	-	-	-	-	Checksum
RunConstFrameServo	N+1	0x10	ICS01-08	ICS09-16	ICS17-24	ICS25-32	ICS33-36	Speed	Position		Checksum
									LOW	HIGH	
~~RunSeriesServo~~	N+1	0x11	ICS01-08	ICS09-16	ICS17-24	ICS25-32	ICS33-36		Speed&Position		Checksum
								Speed	LOW	HIGH	
~~SetSpeed/Stretch~~	N+1	0x11	ICS01-08	ICS09-16	ICS17-24	ICS25-32	ICS33-36	SpSt	Parameters		Checksum
Version	3	0xFD	-	-	-	-	-	-	-	-	0x00
PING	3	0xFE	-	-	-	-	-	-	-	-	0x01
PING(old)	4	0xFE	0x06	-	-	-	-	-	-	-	0x08

Type (for commands 0x00 to 0x0A) is defined as follows:

bit:	7	6	5	4	3	2	1	0	Description	
Type	S	0	Destination		0	0	Source			
							0	0	RAM	Addr = 0x0000 ~ 0x048F
							0	1	ICS	Offset = 0x00 ~ 0xFF, ICS = 0 ~ 35
							1	0	Literal	n = 1 ~ 128 bytes
							1	1	ROM	Addr = 0x000000 ~ 0x03FFFF, size = 1 ~ 128
			0	0					RAM	Addr = 0x0000 ~ 0x048F
			0	1					ICS	Offset = 0x00 ~ 0xFF, ICS = 0 ~ 35
			1	0					COM	
			1	1					ROM	Addr = 0x0000 ~ 0x03FFFF
	0								Send the result to the destination	
	1								Do not send the result to the destination (only update Z and C flags)	

The following rules apply to *Type*:

- MOV is not allowed to use the *S* flag (bit 7).
- Only MOV is allowed to set *Source* to COM (the rest of commands always send a copy to COM).

- Logic, bitwise and arithmetic functions save the result to *Destination* unless the *S* flag is set and their result is calculated as *Destination* = *Destination <operation> Source*.

Size is the number of bytes to use in the operation, the address supplied points to the first byte in the sequence.

Shifts makes a left shift if it is less that 127, and a right shift of (256 – shifts) bits if it is greater or equal to 127.

Conditions is defined as follows:

bit:	7	6	5	4	3	2	1	0	Description
Conditions	0	0	0	0	ON		Match		
					C	Z			If 1, the condition is checked for the Carry and/or Zero flag
							C	Z	Value to match

If *ON* is set, the command will be executed only if the corresponding bits in *Match* are exactly the same as the CPU *carry* and/or *zero* flags.

ICS's *FROM* and *TO* addresses point always to RAM.

In *Const*, *Series* and *Stretch* commands, the ICS bytes are a bitwise selection of which ICS's are being affected. Bit 0 of the first byte is ICS1, bit 1 is ICS2, …, up to bit 3 of the last byte that is ICS36. Following those bytes there is a variable-length list of the configuration of each servo: Position for *Const*, Speed and Position for *Series* and Speed or Stretch for *Stretch*.

The order of the variable-length configuration list is always from the lower ID servo to the highest one selected in the bitwise bytes. For example, if the ICS1, ICS5 and ICS28 bits are set in the bitwise selection bytes, then the variable-length configuration list will be first for the ICS1, then for the ICS5 and lastly for the ICS28 in that order.

The *speed* parameter goes from 1 to 255, being 255 the slowest configuration and 1 the fastest.

10.2.- Appendix B – Expanded hidden structures

Connection structure:

```
struct s_rcb4_connection
{
   int fd; // File descriptor
   fd_set fdset; // Serial file descriptor
   struct termios old_cfg; // Old serial configuration to restore at close
};
```

Command structure:

```
struct s_rcb4_comm
{
   uint8_t size; // Size of the message in bytes
   uint8_t type; // Type of command
   union
   {
      struct s_rcb4_mov
      {
         uint8_t type; // Source and destination. 7th bit not allowed
         rcb4_dst_addr dst;
         rcb4_src_addr src;
      }__attribute__((__packed__)) mov;          // MOVE

      struct s_rcb4_logic
      {
         uint8_t type; // Source and destination. COM not allowed
         rcb4_dst_addr dst;
         rcb4_src_addr src;
      }__attribute__((__packed__)) logic;        // AND, OR, XOR

      struct s_rcb4_not
      {
         uint8_t type; // Source and destination. COM not allowed
         rcb4_dst_addr dst;
         uint8_t zero[2]; // Not used. Must be zero.
         uint8_t size;
      }__attribute__((__packed__)) logic_not;    // NOT

      struct s_rcb4_shift
      {
         uint8_t type; // Source and destination. COM not allowed
         rcb4_dst_addr dst;
         uint8_t zero; // Not used. Must be zero.
         uint8_t shifts;
         uint8_t size;
      }__attribute__((__packed__)) shift;        // RSHIFT, LSHIFT

      struct s_rcb4_math
      {
         uint8_t type; // Source and destination
         rcb4_dst_addr dst;
         rcb4_src_addr src;
      }__attribute__((__packed__)) math;         // +, -, *, /, %

      struct s_rcb4_ics
      {
         uint8_t ics_id;
         uint8_t data_size;
         uint16_t src;
         uint16_t dst;
      }__attribute__((__packed__)) ics;          // ICS

      struct s_rcb4_single
      {
         uint8_t ics_id;
         uint8_t speed;
         uint16_t pos;
      }__attribute__((__packed__)) servo_single; // Single servo command

      struct s_rcb4_const
      {
         uint8_t ics_set[5];
         uint8_t speed;
         uint16_t pos[RCB4_ICS_QTY];
      }__attribute__((__packed__)) servo_const;  // Multiple servos, single speed

      struct s_rcb4_series
      {
         uint8_t ics_set[5];
         struct s_rcb4_speed_pos
         {
            uint8_t speed;
            uint16_t pos;
         }__attribute__((__packed__)) speedpos[RCB4_ICS_QTY];
      }__attribute__((__packed__)) servo_series; // Multiple servos and speeds

      struct s_rcb4_speed
      {
         uint8_t ics_set[5];
         uint8_t spst; // RCB4_SERVO_STRETCH_MODE, RCB4_SERVO_SPEED_MODE
         uint8_t parameters[RCB4_ICS_QTY];
      }__attribute__((__packed__)) servo_speed;  // Speed, stretch
   }command;
} __attribute__((__packed__));
```

10.3.- Appendix C – Documentation errata

The official documentation from KONDO has a few errors that must be addressed. The following lines do not attempt to criticize the documentation but to improve it so that future developers can avoid making the mistakes I did when creating this library. The version being corrected is V220R20140507, precisely the document RCB-4 コマンドリファレンス 20131018.pdf

The first byte received from a successful MOV command to COM is the length of the message and not 04h (page 3). The same for AND, OR and XOR commands (page 5) and arithmetic operations (page 11).

Left shift is for values from 1 to 127 (1 means 1 bit to the left), and right shift is for values from 255 to 128, being 1 shift to the right the value 255 (page 8).

Arithmetic operations do not allow setting the destination to COM (page 10).

Command to move a series of servos with their speed does not work (pages 20, 21 and 22).

The command id to set the speed or stretch conflicts with the one to move a series of servos (page 23) and it says that the speed can only be a value from 1 to 127 while other commands allow values from 1 to 255 (page 24).

11.- References

i http://kondo-robot.com/product/03070
ii http://kondo-robot.com/faq/rcb-4-reference-ver-2_2
iii https://stackoverflow.com/questions/4968529/how-to-set-baud-rate-to-307200-on-linux
iv https://valloric.github.io/YouCompleteMe/
v https://stackoverflow.com/questions/109023/how-to-count-the-number-of-set-bits-in-a-32-bit-integer and http://danluu.com/assembly-intrinsics/