Computarium
Lycée classique de Diekirch
32 av. de la gare L-9233 Diekirch

The 25th Anniversary of the LEGO® RCX®

Claude BAUMANN

Version 1.0
Last edited: July 19, 2023 (19:29)

Abstract

The LEGO RCX (1998) must be considered a milestone in computer technology. This tiny computer was initially designed for kids as a highly sophisticated and yet easy-to-use interface between the real world and a set of LEGO Technic parts –sensors and actuators included– that could be interconnected in an endless number of ways. The goal was to give the kids the opportunity of building and programming toy robots that were able to react and move in function of sensor inputs, and eventually solve real-world problems. 25 years have passed ever since the RCX's first release, and it is time to pay tribute by elevating this real masterpiece of electronic design to the rank of vintage computers. This paper starts with gathering some deep information, in order to allow interested readers –especially vintage computer conservators– to reactivate their BRICK –as the RCX was also called in the community– with the help of modern computers.

History:

- Version 1.0 July 19, 2023

Figure 1: The LEGO RCX with its prototypes and successors, exhibited at the MIT Media Lab in Boston, MA (Photo: CB 2014).

Part I
Foreword

During the writing of this paper, the Adult Fan of LEGO (AFOL) community has been shaken on October 26th 2022 by the announcement of the LEGO Company that the LEGO Mindstorms line will be discontinued at the end of 2022 with app support only guaranteed for another two years. This bad news for all friends of the Mindstorms idea increased even more the authors' motivation for doing this work.

The main goal of the present paper is to uncover some deep and essential information about the RCX from the huge documentation available in books, articles, research papers and last but not least the Internet that should be preserved for posterity –at least for the active museum-conservator of vintage computers.

The paper certainly is not error-free. The author asks the reader who has stumbled over some mistake to give feedback, in order to correct the oversights.

25 years of presence of the LEGO RCX on the Internet has led to an extreme mess of disrupted links, discontinued sites and confusingly mirrored pages, so that many of the web-links listed in this paper have to be checked rather carefully. The COMPUTARIUM holds a stock of this volatile information. So, we invite readers to contact the author via email claude.baumann@education.lu in the case of broken links. Maybe he can help finding the searched document or software.

1 Introduction

There is no doubt: the LEGO RCX (1998) must be considered a milestone in computer technology. Its outstanding position in the series of other legendary devices has many reasons.

First, the RCX was designed for kids as a highly sophisticated toy, and yet an easy-to-use interface between the real world and a set of LEGO Technic bricks –sensors and actuators included– that could be interconnected in a literally endless number of ways. The goal was to give the kids the possibility of building and programming tiny robots, which were able to react and move in function of sensor inputs, and eventually solve real-world problems.

Second, the RCX as being part of the revolutionary Mindstorms concept, rapidly found its way into schools and even universities as a perfect educational tool. Undoubtedly, by fostering the idea of problem solving using computers and networks, LEGO robotics played a pioneering role in realizing late MIT professor Seymour Papert's vision of constructivist STEM education.[1]

Third, the LEGO Robotics Invention™ kit, with the RCX as the intelligent brick, was innovative in many ways. It made use of the newly accessible World Wide Web, in this vein sustaining the emergence of fan-communities of all ages all over the world. Also, the RCX was one of the very first devices to be graphically cross-programmed on ordinary home-computers.

25 years have passed since the RCX's first release and its tremendous success, which lasted for an incredibly long duration in terms of computer life-times. This prolific period only ended with the release of its successor, the LEGO NXT in 2006 and a few years later the LEGO EV3 in 2013. Sadly, most of the RCX bricks now have ended on the attic. However, after a quarter century, the RCX has joined the rank of worthy old-timers whose secrets should not be forgotten.

[1] S. Papert, *Mindstorms, Children, Computers and Powerful Ideas*, Basic Books Inc., NY, (1980).

2 Robot “GASTON”, the World’s most complex LEGO RCX robot

A vintage LEGO Mindstorms robot is owned by the COMPUTARIUM (cf. Fig. 2). The humanoid robot was built and programmed in 2003 by a group of students of the Convict Episcopal boarding institution, Luxembourg.[2] It uses two RCX controllers and a bunch of sensors and motors. Counting among the most elaborated LEGO robots, “GASTON” was invited to the 20th anniversary of Lego Mindstorms during September 2018. This was a special exhibition held at the LEGO HOUSE in Billund, Denmark (cf. Fig. 3).

“GASTON”’s actual status is ‘nonfunctioning’. Some of the original LEGO sensors have rotten wires, which seems to be a typical behavior of the very first generation LEGO Mindstorms material (Fig. 4). All of these wires needed replacement, requiring a complete revision of “GASTON”’s setup.[3] Fortunately, the builders had thought of drawing CAD files of each building step using third party software.

Initially, “GASTON” was programmed using the ROBOLAB environment (created by the CEEO at Tufts University, Boston). Unfortunately ROBOLAB is difficult to run on actual computers, as the software was discontinued beyond 2013.[4] Note that “GASTON” required pre-installed LEGO RCX firmware (version 3.28), on which “GASTON”’s multi-tasking programs were superimposed.

GASTON

LCD
Lycée classique de Diekirch

Fabricant	CONVICT LU
Nom	
Modèle	GASTON
Année	2003
Type	Robot LEGO
Processeur	RCX 8bit H300/8
Ram	32 kB
Stockage	
Misc.	2 modules RCX
Database	807
Fonctionne	non
Utilisé LCD	non

Robot LEGO développé par des étudiants du Convict de Luxembourg et leur directeur C. Baumann. Basé sur Lego Mindstorms. Réagit sur des informations sonores, lumineuses et de température. Faculté d’expression fasciale. Programmation complexe en Labview. (Photo Convict). Don Claude Baumann (2014).

http://www.convict.lu/Jeunes/Gaston/User_guide.htm

Figure 2: Computarium file of robot “GASTON”.

[2] cf. *Sticky* section of https://www.convict.lu/index_r.php, [retrieved 10,2022].
[3] https://www.youtube.com/watch?v=hV13i88nPVM, [retrieved 11/2022].
[4] https://ceeo.tufts.edu/, [retrieved, 10.2022].

Figure 3: "GASTON" at the LEGO Mindstorms 20th anniversary exhibition at LEGO HOUSE (Billund, DK).

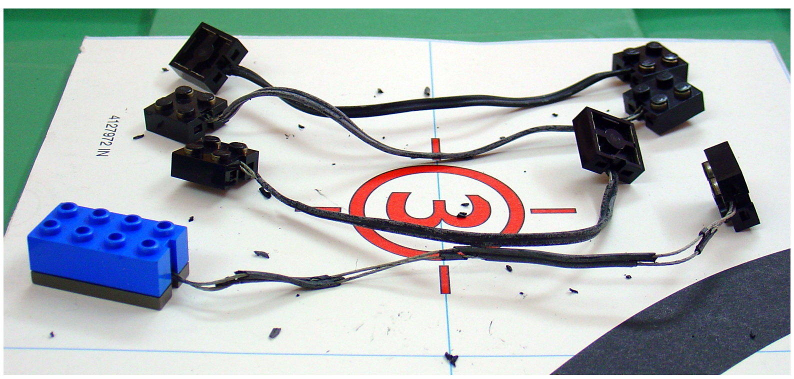


Figure 4: Rotten LEGO Mindstorms wires.

Part II
"Hello World"

This document part presents a few valuable hints on how to revive the RCX with modern computers.

3 Getting started

3.1 Installing the LEGO IR-Tower with MS Windows

Normally the LEGO RCX is programmed using the LEGO IR-Tower, which exists in two versions: RS232 and USB. Note that actual MS WINDOWS versions do no longer support the USB Tower. By contrast, the serial tower can be successfully used with a modern USB-RS232 adapter. With some patience it should be possible

to install and run the third party BRICKCC software that can be downloaded from:
https://sourceforge.net/projects/bricxcc/files/bricxcc/.

Some excellent hints on how to reactivate the RCX can also be found at the following web pages. Note that most of these sites recommend installing WINDOWSXP on a VIRTUALBOX.[5]

- https://lehubbycodershah.blogspot.com/p/rcx.html?m=1
- https://www.bartneck.de/2017/06/08/using-your-lego-mindstorms-rcx-on-a-modern-computer/
- https://www.johnholbrook.us/RCX_guide.html
- https://www.eurobricks.com/forum/index.php?/forums/

The most important information about the RCX internals can be found at:

- Famous Kekoa Proudfoot reverse engineering page: http://www.mralligator.com/rcx/
- Mirror site: https://www.cs.montana.edu/courses/spring2005/445/resources/downloads/RCX/
- Other mirror site https://www.tech-insider.org/lego-mindstorms/research/1999/0429.html
- Basic information: https://www.classes.cs.uchicago.edu/archive/2006/fall/23000-1/docs/rcx.pdf
- Software Development Kit (SDK) https://www.philohome.com/sdk25/sdk25.htm

3.2 Installing the USB Tower on a Raspberry Pi

The following method observes the instructions that can be found at:
https://minordiscoveries.wordpress.com/.

Plug in the USB Tower : In the RPi terminal type the command:

```
find /lib/modules -name *lego*
```

This should produce a comparable result:

```
/lib/modules/3.10.18+/kernel/drivers/usb/misc/legousbtower.ko
/lib/modules/3.10.25+/kernel/drivers/usb/misc/legousbtower.ko
```

Create a rule for the USB device : This will allow anybody in the group 'lego' to have access to it.

First create the following file: `/etc/udev/rules.d/90-legotower.rules` by typing the command line:

```
sudo nano /etc/udev/rules.d/90-legotower.rules
```

This opens the specified file in the editor `nano`. Now add this single line to the file and save it with `CTRL+X` and close `nano`.

```
ATTRS{idVendor}=="0694",ATTRS{idProduct}=="0001",MODE="0666",GROUP="lego"
```

Create a lego group for the device : (assumed you are the user 'pi'.)

```
sudo groupadd lego
sudo usermod -a -G lego pi
```

[5] https://www.virtualbox.org/.

3.3 Installing Dave Baum/John Hansen's NQC on the Raspberry Pi

Exactly follow the instructions shown at: https://minordiscoveries.wordpress.com/.

Download NQC : (The NQC software is known worldwide for being the best C++ environment for the RCX.)

```
mkdir nqc-3.1.r6
cd nqc-3.1.r6
wget http://bricxcc.sourceforge.net/nqc/release/nqc-3.1.r6.tgz
tar xfz nqc-3.1.r6.tgz
cd ..
```

Follow the detailed instructions listed on the cited web-site : The instructions are very clear.

HOWEVER: before typing the command `sudo make install`, you will need to erase a single line in the `SRecord.cpp` file in the `/nqc-3.1.r6/nqc` directory at the end of the `int srec_decode(srec_t *srec, char *_line)` function:

```
sum += C2(line, pos);

~~if ((sum & 0xff) != 0xff)~~
    ~~return SREC_INVALID_CKSUM;~~

return SREC_OK;
```

Note that these changes are required, in order to allow NQC to download any non-LEGO firmware to the RCX.

Now continue NQC install procedure : You may of course use NQC for programming the RCX in C++.

Download any valid firmware to the RCX : This is done by typing (assumed the firmware is in the current directory):

```
nqc -Susb:/dev/usb/legousbtower0 -firmware NAME.srec
```

Apparently the USB Tower specification in this command line is necessary to access the device.

3.4 Homebrew IR-Tower controlled with the Raspberry Pi

As an alternative, we propose to replace the tower by elementary circuitry (Fig. 5), which uses the Vishay TSOP1738 (or any similar IR receiver module for remote control systems, Fig. 6) with output active low, i.e. where the output goes down to 0V in the presence of an 38kHz IR-signal. The circuit also requires an IR-LED 940 nm (for instance Vishay TSAL 6200).

Both transistors Q1 and Q2 form a logical AND gate. Q1 is controlled by a 38kHz signal, which is produced by the RPi hardware PWM (GPIO 1 or Pin 12). Q2 is controlled by TxD (Pin 8). The UART device of the RPi is active low, so the TxD MARK/STOP = 3V3, while START/LOGICAL 1 = 0V. Thus, Q2 becomes conductive only with START/LOGICAL 1.

The output of TSOP1738 is directly wired to RPi RXD (Pin 10).
Power is supplied through Pin 4 (5V) and GND (Pin 6).

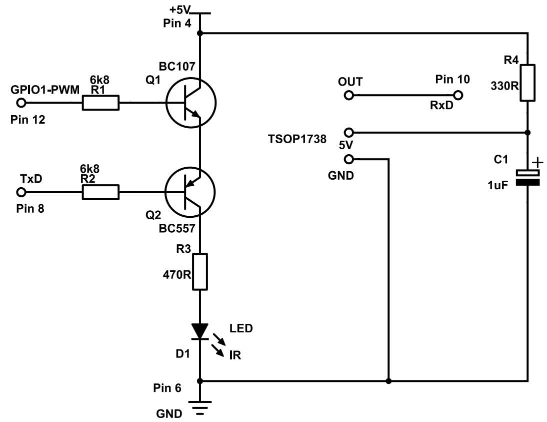


Figure 5: Elementary circuitry required for this project. (cf. Fig. 7)

Figure 6: IR receiver module.

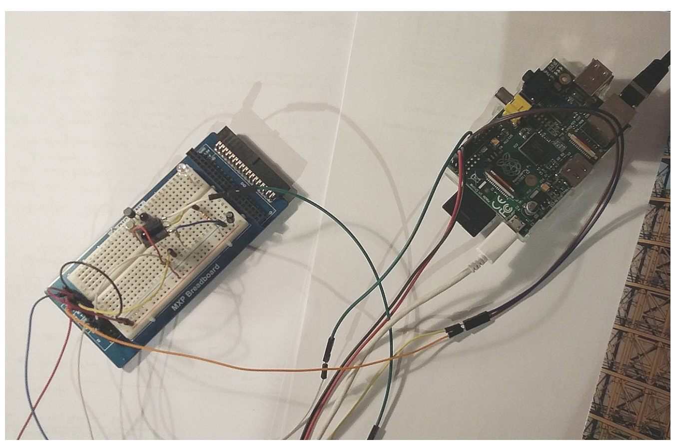


Figure 7: Prototype.

3.5 Software download with the homebrew IR-Tower

The actual software `RCX_download.c` that is needed here can be downloaded from:
https://github.com/pnc/rcx/blob/master/RCX_Download.c
This program is mostly based on Kekoa Proudfood's 1999 download software.[6]

A few lines have either to be changed or added in the original program.

[6]Available online: https://github.com/michaelko/tvm/blob/master/tools/tvm_firmdl3.c, [retrieved November 2022].

The RPi needs some preparation:

Raspbian :

The RPi should run under the RASPBIAN environment provided by the NOOBS package available at http://www.raspberrypi.org/downloads.

WiringPi :

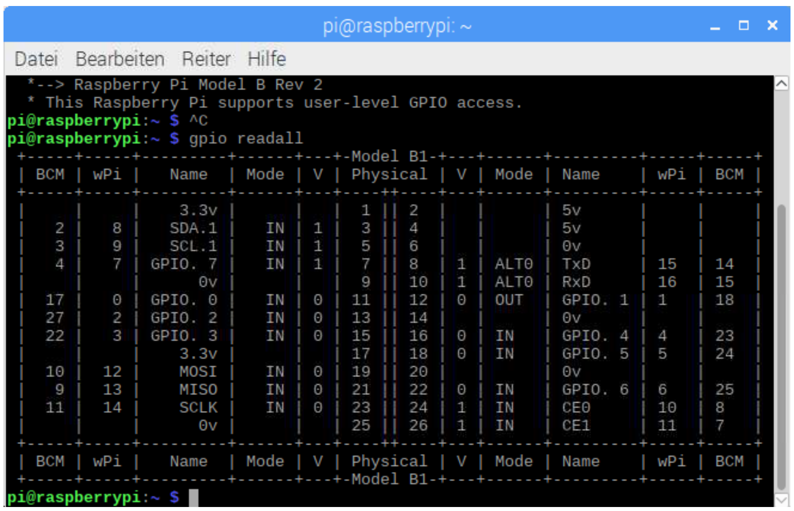
In order to control the RPi hardware PWM, which is needed to generate a 38kHz carrier for the IR signals, the WIRINGPI software package must be installed on the RPi (http://wiringpi.com). This can be done easily in the terminal window through:

```
sudo apt -get install git - core
git clone git://git.drogon.net/wiringPi
cd wiringPi
./build
```

Test the correct install through `gpio -v`, which should give something alike:

```
gpio version:  2.46
Copyright (c) 2012-2018 Gordon Henderson...
```

The `gpio readall` command returns the current pinout of the RPi (cf. Fig. 8).



```
pi@raspberrypi: ~
Datei  Bearbeiten  Reiter  Hilfe
  *--> Raspberry Pi Model B Rev 2
  * This Raspberry Pi supports user-level GPIO access.
pi@raspberrypi:~ $ ^C
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+-Model B1-+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5v      |     |     |
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 1 | ALT0 | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | ALT0 | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | OUT  | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+-Model B1-+---+------+---------+-----+-----+
pi@raspberrypi:~ $
```

Figure 8: RPi terminal return on `gpio readall` command.

Changes to apply to RCX_download.c :

1. Add line
   `#include <wiringPi.h>`
   to the program header.
2. Add a single semicolon ';' after the `default:` case of the `switch` instruction in the `void print_answer(answer a)` function. The RPi GCC compiler doesn't accept the original code.
3. Change line in the /* RS232 ROUTINES. */ section of the program
   ~~`#define DEFAULT_RCX_IR "/dev/term/a" /* Solaris name of serial port */`~~
   to
   `#define DEFAULT_RCX_IR "/dev/ttyAMA0" /* RPi1 B name of serial port */`

4. Inside the `int IR_open()` function, between the lines:

```
if (tcsetattr(fd, TCSANOW,& ios) == -1) {
    perror("tcsetattr");
    exit(1);
}

return fd;
```

   add the following PWM control instructions:

```
if (tcsetattr(fd, TCSANOW,& ios) == -1) {
    perror("tcsetattr");
if(wiringPiSetup()==-1) printf("WiringPi not installed");
pinMode(1,PWM_OUTPUT);
pwmSetMode(PWM_MODE_MS);
pwmSetClock(21);
pwmSetRange(24);
pwmWrite(1,12);
    exit(1);
}

return fd;
```

   This additional code activates the hardware PWM at 38kHz with 50% duty cycle.

   Note that the RPi system frequency 19.2E6Hz available for the PWM must be divided by 505.26≈21*24 in order to obtain 38kHz. The duty cycle is 50%. (The internal counter changes the PWM's state after having counted 12.)

   Also note that the PWM is GPIO 1, while physically being Pin 12.

   Finally note that the PWM must be run under the MARK/SPACE mode instead of the default BALANCED mode.

5. Anywhere inside the `void IR_close(int fd)` function add the following lines:

```
pwmWrite(1,0);
pinMode(1,OUTPUT);
```

   which switches off the PWM, while the serial connection is being closed.

6. Once saved, the program is ready to be compiled. This is done in the RPi terminal through:

```
gcc -I . -o RCX_RPi /home/pi/Dokumente/my_c/RCX_download_rpi.c -l wiringPi
```

   Now any valid RCX firmware can be downloaded by typing:

```
sudo ./RCX_RPi ./NAME.srec
```

4 Dead RCX

The RCX has been produced in three versions 1.0, 1.5 and 2.0. Some damages have been reported for all three versions. Hints may be found in the major discussion forums:

- https://news.lugnet.com/robotics/
- https://brickshelf.com/cgi-bin/gallery.cgi?f=38746
- https://www.youtube.com/watch?v=Y2BG48_HHy8

- https://www.youtube.com/watch?v=O-_kaDSuFYQ
- https://www.youtube.com/watch?v=hV13i88nPVM
- https://www.youtube.com/watch?v=E8Do_jUyMQk
- https://www.youtube.com/watch?v=MAQTOPznZ6U

There are four major categories of damages:

1. Rotten sensor and motor wires (cf. Fig.4)
2. RCX 1.0 with AC power adapter: voltage regulation damaged because of excessive current (diodes, regulator, fuse)
3. All versions: PC board damaged due to leaking batteries
4. All versions: IR-LED circuitry damaged because of excessive use of long range (LEDS, diodes, transistors)

5 Writing "Hello World" to the RCX display

Now that the reader should be able to download **firmware** to the RCX, he or she could of course start playing with the original software. However, as it is the purpose of this paper to gather some essential secrets for interested conservators of historic computers, we'd like to plunge more deeply into the subject. In order to learn more about the RCX internals, we reproduce the code of an elementary RCX firmware that correctly initializes the device with a few basic functions, writes "Hello World" to the display, while waiting two seconds between the screens and then return to the RCX **executive** (cf. Fig. 9). The reader will certainly have noted the two keywords that we are going to explain in the following paragraphs.

hello_world.srec


```
S01300003F4C49425F56455253494F4E5F4C303046
S1138000790101F41B01790202941B020D2246FA44
S11380100D1146F07900CC407901F0005E00043681
S11380205E0080C4FEFF6A8ECC8BFEFF6A8ECC8C11
S11380305E0083C6790600006B86CCA2FE566A8E6B
S1138040CC86FE646A8ECC87FE646A8ECC88FE671A
S11380506A8ECC89FE726A8ECC8A5E00826A5E0069
S113806027C8790500007906019 05E0084707906BE
S11380700006B86CCA2FE5E6A8ECC86FE646A8E9D
S1138080CC87FE426A8ECC88FE566A8ECC89FE7CF2
S11380906A8ECC8A5E00826A5E0027C87905000079
S11380A0790601905E0084700480790101F41B015B
S11380B0790202941B020D2246FA0D1146F05F006C
S11380C05A0080C0FE016A8ECC40FE006A8ECC410C
S11380D0FE016A8ECC42FE036A8ECC43FE036A8E96
S11380E0CC44FE016A8ECC45FE006A8ECC46FE016D
S11380F06A8ECC47FE036A8ECC48FE036A8ECC4956
S1138100FE016A8ECC4AFE006A8ECC4BFE016A8E5B
S1138110CC4CFE036A8ECC4DFE036A8ECC4EFE021F
S11381206A8ECC4FFE046A8ECC50FE056A8ECC510B
S1138130FE086A8ECC52FE076A8ECC53FE026A8E0C
S1138140CC54FE046A8ECC55FE056A8ECC56FE08CE
S11381506A8ECC57FE076A8ECC58FE046A8ECC59C1
S1138160FE056A8ECC5AFE066A8ECC5BFE086A8ECA
S1138170CC5CFE076A8ECC5DFE046A8ECC5EFE0587
S11381806A8ECC5FFE066A8ECC60FE086A8ECC6176
S1138190FE076A8ECC62FE106A8ECC63FE106A8E76
S11381A0CC64FE016A8ECC65FE016A8ECC66FE103D
S11381B06A8ECC67FE206A8ECC68FE206A8ECC69FC
S11381C0FE026A8ECC6AFE026A8ECC6BFE206A8E39
S11381D0CC6CFE806A8ECC6DFE806A8ECC6EFE08FF
S11381E06A8ECC6FFE086A8ECC70FE806A8ECC716C
S11381F0FE206A8ECC72FE026A8ECC73FE026A8EF9
S1138200CC74FE026A8ECC75FE026A8ECC76FE803B
S11382106A8ECC77FE086A8ECC78FE086A8ECC799C
S1138220FE086A8ECC7AFE086A8ECC7BFE206A8EAD
S1138230CC7CFE206A8ECC7DFE206A8ECC7EFE2017
S11382406A8ECC7FFE206A8ECC80FE806A8ECC81C4
S1138250FE806A8ECC82FE806A8ECC83FE806A8E1D
S1138260CC84FE806A8ECC855470790600056B86BC
S1138270CCA8790500006B06CCA81D5642045A0012
S113828083C41B066B86CCA8FE406A8ECCAA7906F4
S11382900076B86CCAC790500006B06CCAC1D5692
S11382A042045A0083C01B066B86CCAC6B00CCA880
S11382B07901CC860910680E6A8ECCAE6A0DCCAA02
S11382C06A0ECCAE16DE6A8ECCAE6B06CCAC6B867A
S11382D0CCB0790500056B06CCB05E0001306B8630
S11382E0CCB07905CC406B06CCB009566B86CCB0CD
S11382F06B06CCA86B86CCB26B05CCB06B06CCB24D
S113830009566B86CCB26B00CCB2680E5E0084DC81
S11383106B86CCB47905EF436B06CCB409566B86FA
S1138320CCB46B06CCAC6B86CCB0790500056B0682
S1138330CCB05E0001306B86CCB07905CC636B06A6
S1138340CCB009566B86CCB06B06CCA86B86CCB290
S11383506B05CCB06B06CCB209566B86CCB26B0008
S1138360CCB2680E6A8ECCB6FD006A0ECCAE1CDEB5
S113837047045A0083A06A0ECCB6170E6A8ECCB69B
S113838006A0DCCB66B00CCB4680E16DE6B00CCB4B3
S11383906 88E6A0ECCB6170E6A8ECCB65A0083B2BE
S11383A06A0DCCB66B00CCB4680E14DE6B00CCB495
S11383B0688E6A0ECCAA110E6A8ECCAA5A008296D9
S11383C05A00827254707906 00A56B86CCB8790682
S11383D000016B86CCBA790600016B86CCBC547067
S11383E06B06CCB86B86CCBE6B05CCBC6B06CCBE29
S11383F05E0001306B86CCBE7905000A6B06CCBEEF
S11384005E0001306B86CCBE6B05CCBA6B06CCBE71
S11384105E0001BE6B86CCBE790500056B06CCBE46
S113842009566B86CCBE7905000A6B06CCBE5E0091
S113843001BE6B86CCBE79057FFF6B06CCBE1D5698
S1138440430879067FFF6B86CCBE790500016B0679
S1138450CCBE1D5644087906 00016B86CCBE6B0667
S1138460CCBE6B86CCB86B06CCBA6B86CCBC5470D9
S11384707903000079 04000A5E0001FE5E0084E0DA
S11384805E008486547079000006 19076FF60000BC
S11384907906 00006FF600026F7500006F7600022B
S11384A01D5645045A0084D40B066FF60002790667
S11384B000006FF600046B05CCB86F7600041D5603
S11384C045045A0084D00B066FF600045A0084B6A7
S11384D05A008498790000060907 5470F600547019
S11384E07375470879050000790600005470000094
S11384F0446F20796F7520627974652C20776865E8
S11385006E2049206B6E6F636B3F000000000000020
S90380007C
```

Figure 9: Improvised letters on a rudimentary LCD-display. “World” follows on a second screen.

Part III
Fundamentals

6 What is an RCX firmware?

Normally, a firmware is a sort of computer software that is stored in a non-volatile memory space. Its role is, roughly said, to make the system run, at least at a low level, so that user programs can be started with its help. In fact, the RCX has a built-in program code that corresponds to this definition. It is burned in a ROM section of the Hitachi (Renesas) H8/3292 micro-controller, which represents the heart of the LEGO Brick. Interestingly, the LEGO slang calls this program part the RCX executive, whereas the firmware is a RAM-based operating system that is downloaded from the PC via IR-Tower. Managing the download process is the main task of the ROM-executive.

Official and unofficial LEGO RCX firmware can be downloaded from:
https://pbrick.info/index.html-p=74.html

Note that the extension for official firmware always is **.lgo**. The download software that is part of the official cross-programming environment only accepts firmware with this extension. Also, the code must be packed into the S-record format.

6.1 Motorola S-record Format

Usually RCX firmware data are wrapped in the widely used S-record format that was first applied with the Motorola 6800 processor.[7] RCX-relevant S-records **(srec)** consist of a sequence of ASCII character strings:

Example:
S01300003F4C49425F56455253494F4E5F4C303046
S1138000790600076B86CC000000446F20796F75F9
S113801020627974652C207768656E2049206B6E28
S11380206F636B3F000000000000000000000000D0
S9038000 7C

[7] https://en.wikipedia.org/wiki/SREC_(file_format), [retrieved 11.2022].

Legend:

1. S0: record type : address field unused and filled with zeroes; data-field = header information. (In the current example, the cross-compiler, while creating the firmware, added the particular ASCII code line representing the text: *?LIB_VERSION_L00*).
2. S1: address field is interpreted as a 2-byte address; the data field is composed of memory loadable data
3. S9: address field contains the starting execution 2-byte address (irrelevant here, since the RCX executive always starts the firmware at 0x8000)
4. nn: number of following bytes in hex-notation (0x13 $=19_{10}$, thereof 16 firmware payload bytes.)
5. xxxx: address where to store record
6. ssssss: payload data
7. cc: checksum of data bytes in the record. (Calculating the checksum obeys the following rules for S1 records):
   - Only consider the address and the data part of the record.
   - Clear the highest bit in the address =Addr & 0x7FFF.
   - Byte-wise add the byte values in byte representation, while ignoring any overflow.
   - Add the constant value 0x93 to the result.
   - The final result is obtained by taking the 2s complement.

   Note: This method does not work for the S0 and S9 records. (The obscure algorithm for the "checksum" of those records has not been reversed engineered. That's why unofficial firmware simply uses copies of the original S0 and S9 records, which does not affect the firmware encoding, since no relevant data are stored in these records.)

7 Firmware download protocol

The download software extracts the payload data from the S-record file and recombines them into a new packet form that is suitable for being sent via infrared channel. Because infrared transmission must be considered as an insecure communication path from the communication theory point of view, the download software must implement a few security measures protecting the data from being uselessly altered. The RCX executive must of course be able to decrypt this encoding, in order to extract the original data with 100% certainty.

7.1 UART

We learned so far that the IR-channel works with a 38kHz signal. Data is sent using the regular asynchronous UART protocol with the following settings:

- MARK=38kHz IR-signal off, SPACE=IR-signal on
- 2400 baud
- Odd parity
- 1 Stop bit

The transmission may be disturbed by flickering light sources, such as high frequency Neon-tubes. However, the most important disturbances come from the IR-channel itself. Both, the cross-programming PC with the IR-tower and the RCX receive their own transmission echoes at hardware level, because the UART hardware works in full duplex mode. Therefore each participating device has to reduce the traffic to half duplex by software means. In other words, both of them must ignore received messages during their own transmission activity.

7.2 Data packets

RCX messages are packed into secure data packets.

Example of a valid packet directed to the RCX:

55 FF 00 65 9A 01 FE 03 FC 05 FA 07 F8 0B F4 80 7F

RCX reply to this message:

55 FF 00 9A 65 9A 65

7.2.1 55 FF 00 Header

In order to activate the IR-channel and signalize the beginning of a new IR-message, every packet must necessarily start with a 0x55 FF 00 header.[8] The 0x55 byte corresponds to its bit representation b'01010101'. This makes a complete IR-signal 1010101010(0), start, parity and stop bits included. Although no signal synchronization is performed by the ROM executive, this initialization byte is well understood by the RCX, because of its balanced appearance.

7.2.2 Data byte and and its 2s complement

The principle of balancing is continued by the sending of every data byte as a pair with its 2s complement. The executive verifies the correctness of each pair. In the case of a failure, the RCX RX interrupt handler rejects the packet and the ROM executive either sends an error reply or no reply at all, inviting the cross-programming device to repeat the packet sending.

7.2.3 Checksum

The checksum is the simple byte sum of all the data bytes. (The header is not counted.)

7.3 Opcodes

RCX messages always start with an opcode command byte. For instance, the most simple opcode is the **Ping** command 0x10. The RCX software designers have opted for the following opcode structure:

`X X X X T N N N`, where

- X X X X is the opcode's first nibble
- T is the toggle bit, which is always used, if the same opcode is sent successively. For example, repeated pinging will get the following packet pattern:

  55 FF 00 10 EF 10 EF
  55 FF 00 18 E7 18 E7

[8]Although it seems that packets might be as well received without the 0x55 byte.

55 FF 00 10 EF 10 EF
55 FF 00 18 E7 18 E7...

- N N N indicates the number of parameters to expect. (*For program size efficiency, the unused lengths 6 and 7 actually mean 0 and 1 respectively. This measure doubles the number of available commands with few parameters.*[9])

7.4 Firmware download sequence

1. Ping: (see if the RCX is alive) **0x10** / RCX reply: **0xEF**
2. Reset: (send the RCX into boot mode) **0x65, 1, 3, 5, 7, B** / RCX reply: **0x92**
   (N.B.: Commas are not sent, they are used here as separators.)
3. Begin download: (check, if there is enough memory space) **0x75, Start address (LO), Start address (HI), Firmware checksum (LO), Firmware checksum (HI), 0** / RCX reply: **0x82**
   (NB: *The firmware checksum is calculated as the byte sum of the first 19456 bytes in the firmware program (*$= 19 \cdot 1024 = 19K \mod 65536$*. If the size of the firmware file is less than 19 K, zeroes are assumed for the remainder.*[10])
4. Download: (the firmware is sent in blocks of N bytes, where N=200 seems a good compromise between packet duration and send retries due to transmission errors.) **0x45, Block number (LO), Block number (HI), N=Number of bytes in this block (LO), N (HI), Data byte [0], Data byte [N-1], Block checksum**
   (N.B.: Blocks are numbered 1..M, 0. The very last block is numbered 0. This tells the RCX that no block will follow.)
   The RCX reply is **0xB2, Status**, where Status is:
   - 0: OK
   - 3: Block checksum error
   - 4: Firmware checksum error
   - 6: RCX not in boot mode

   Two hand-shake issues may be observed here:

   - If the RCX received the block without error and therefore sent the 0xB2 reply with Status=0, it might happen that the reply gets lost. In that case the PC thinks the packet was corrupted and tries to send it again. Because the RCX expects the block with the incremented ID-number but gets the wrong one, it sends an error reply. Now the PC assigns this reply to the old block, and the download process gets hooked.
   - If the last block (ID-number 0) is not well received by the RCX, the firmware download process is aborted. Retries from the PC are not accepted.
5. Unlock the firmware: (checks the firmware integrity and unlocks it) **0xA5, 4C 45 47 4F AE = 0xA5, 'L' 'E' 'G' 'O' '®'** / RCX reply: **0x52, "Just a bit off the block!"**.

   Important note: The firmware absolutely needs to end with the ASCII values of the text: **"Do you byte, when I knock?"**. For any homebrew firmware, it is essential to make sure that the very last byte before this text hasn't the value 0x44 (=ASCII-code for the letter D), because otherwise the ROM executive will start verifying the unlock text one byte too early with certain failure, and the firmware won't be unlocked.

[9] SDK, Firmware overview, cf. link list in section 3.1
[10] SDK, p.92.

8 ROM executive

So far, it must have become clear that the ROM executive's most important role is to act as a boot-loader. However, besides this task, the executive also handles button events, which activate some elementary motor functions and sensor reading tasks, certainly used for factory quality check. It also controls the battery survey, the RCX display and the sound generator. In order to do all this, the ROM executive must have correctly initialized all hardware modules and interrupts. It runs as a single state machine in the main task, managing all of the required RCX functions for the boot-loading and port testing processes.

9 LEGO Assembly Mnemonics (LASM)

The RCX designers did a great job in designing the original firmware. The difficulties must have been gigantic, because the firmware's priory function consisted in running a real robot operating system far beyond the limited capacities of the ROM executive. This should allow several robot state machines to run in parallel along with the system and the interrupt handlers. Here a non-exhaustive list of firmware missions:

- System initialization and shut-down
- Sensor reading
- Battery survey and power management
- Display update (For instance, keep the little running man moving)
- UART received opcode interpretation and reaction
- Button survey
- Motor state update
- Sound output control
- Management of multi-tasking
- Execution of user programs

Each of these missions has its own complexity, which is increased by the fact that the relevant code execution has to run in harmony with the interrupt handlers while maintaining the robustest system stability, and all this within the bounds of very limited memory space and clock speed. In no way this stability should be affected by user programs. And vice-versa, the background system should never disturb user program execution, because undesired and unpredictable robot behavior might otherwise emerge making debugging almost impossible, especially for kid users.

The LEGO engineers invented the LASM commands as small chunks of code –very close to the H8/3292 Assembly language– that could be downloaded and executed at the highest possible speed, and yet consume minimal memory space. The LASM commands and their syntax were crystal clear and easy to handle both for the experienced programmer and higher level compilers used by the kids.

Note that third party software, although pushing the RCX to its limits, didn't necessarily care about balanced execution or system stability. Also, badly used, such software could damage one or the other electronic part of the RCX. At least, more than one destroyed IR-LED driver circuits can be credited to inadequate application of third party software.

10 Single task firmware

The following code snippet represents a fully working firmware with a single function of waiting for any button pressed event. The states of the buttons are polled in the unique main routine. If the corresponding byte-value isn't zero anymore, the RCX is reset. Note that after downloading this software, the characteristic fast rising sweep sound is being heard indicating that the firmware was successfully unlocked. And then,... nothing happens unless the user presses one of the buttons, all of which produce the same effect. Also note that the program does not alter the elementary initialization performed by the ROM executive before the firmware download. **Important note:** In the following lines we will present some H8/3292 Assembly code. If the reader wants to compile and assemble his own code, he or she will need a cross-compiler environment for the H8. This can be found at https://www.cs.scranton.edu/ bi/brickos/brickos.htm for instance.

test_buttons.asm / test_buttons.srec

```
label begin_of_program
//WAIT 500ms
mov.w #0x1F4,r1
label L_sys_1002
subs #0x1,r1
mov.w #0X294,r2
label L_sys_1003
subs #0x1,r2
mov.w r2,r2
bne L_sys_1003
mov.w r1,r1
bne L_sys_1002

label begin_of_task_0
//clear user memory
mov.w #0xcc40,r0
mov.w #0xF000,r1
jsr @0x436
jsr sys_read_buttons
label beginloop_1004
//IF RCX Button states(0) = 0
mov.w #0x0,r5
mov.w @0xEE28,r6
cmp.w r5,r6
//  =
beq loopdo_1004
jmp endloop_1004
label loopdo_1004
jsr sys_read_buttons
jmp beginloop_1004
label endloop_1004
//reset RCX
jsr @@0x0
label end_of_task_0

//if we ever came here
jmp end_of_task_0
label sys_read_buttons
//ROM_CALL
mov.w #0xEE30,r6
mov.w r6,@-r7
mov.w #0x4000,r6
jsr @0x29F2
adds #0x2,r7
// @0XEE2C = @0XEE30
mov.w @0xEE30,r6
mov.w r6,@0xEE2C
//ROM_CALL
mov.w #0xEE2A,r6
mov.w r6,@-r7
mov.w #0x3000,r6
jsr @0x1FB6
adds #0x2,r7
mov.w @0xEE2C,r6
shlr r6L
xor #0x1,r6L
shll r6L
shll r6L
shll r6L
mov.w @0xEE2A,r4
or r4L,r6L
mov.w r6,@0xEE2C
// @0XEE28 = @0XEE2C
mov.w @0xEE2C,r6
mov.w r6,@0xEE28
rts 0
label end_of_program
```

```
S01300003F4C49425F56455253494F4E5F4C303046
S1138000790101F41B01790202941B020D2246FA44
S11380100D1146F07900CC407901F0005E00043681
S11380205E008042790500006B06EE281D56470469
S11380305A00803C5E0080425A0080245F005A004F
S1138040803E7906EE306DF6790640005E0029F236
S11380500B876B06EE306B86EE2C7906EE2A6DF6F6
S1138060790630005E001FB60B876B06EE2C110EEE
S1138070DE01100E100E100E6B04EE2A14CE6B8669
S1138080EE2C6B06EE2C6B86EE2854700000446FC9
S113809020796F7520627974652C207768656E206D
S11380A049206B6E6F636B3F00000000000000000E
S90380007C
```

11 H8/3292 Micro-controller

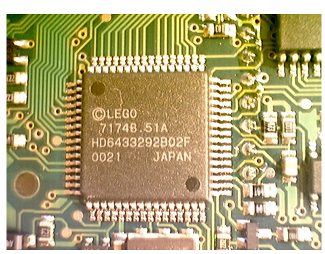


Figure 10: View on the H8/3292 microprocessor that controls the RCX.

Datasheet for the H8/3292 can be downloaded from the following sites:

1. https://docs.rs-online.com/bd6a/0900766b8002614f.pdf
2. https://www.cs.scranton.edu/ bi/2007s-html/cs358/hitachi.pdf

As already said, the heart of the RCX is a Hitachi (Renesas) H8/3292 micro-controller. It is clicked at 16MHz. The features are:

CPU: (H8/300 core)

- Eight 16-bit registers r0 .. r7, or Sixteen 8-bit registers r0H, r0L, .., r7H, r7L
  r7 is used as the stack-pointer.
- 16-bit program-counter (PC)
- 8-bit condition code register (CCR)
- Maximum clock rate: 16 MHz at 5 V
- 8- or 16-bit register-register add/subtract: 125 ns (at 16 MHz)
- 8*8-bit multiply: 875 ns (at 16 MHz)
- 16/8-bit divide: 875 ns (at 16 MHz)
- Concise instruction set, instruction length: 2 or 4 bytes
- Register-register arithmetic and logic operations
- MOV instruction for data transfer between registers and memory

Memory:

- 16k-byte ROM; 512-byte RAM
- Operating modes:
  - Expanded mode with on-chip ROM disabled (mode 1)
  - Expanded mode with on-chip ROM enabled (mode 2)
  - Single-chip mode (mode 3)

16-bit free-running timer (1 channel):

- One 16-bit free-running counter (can also count external events)
- Two output-compare lines
- Four input capture lines (can be buffered)

8-bit timer (2 channels):

- One 8-bit up-counter (can also count external events)
- Two time constant registers

Watchdog timer (1 channel):

- Overflow can generate a reset or NMI interrupt
- Also usable as interval timer

Serial communication interface (SCI) (1 channel):

- Asynchronous or synchronous mode (selectable)
- Full duplex: can transmit and receive simultaneously
- On-chip baud rate generator

A/D converter (ADC) (8 channels):

- 10-bit resolution
- Single or scan mode (selectable)
- Start of A/D conversion can be externally triggered
- Sample-and-hold function

Interrupts:

- 4 external interrupt lines: $\overline{\text{NMI}}$, $\overline{\text{IRQ0}}$ to $\overline{\text{IRQ2}}$
- 19 on-chip interrupt sources

Interrupt	Description	Vector table address
$\overline{\text{NMI}}$	Non-maskable interrupt	0x0006
$\overline{\text{IRQ0}}$	External interrupt request	0x0008
$\overline{\text{IRQ1}}$	External interrupt request	0x000A
$\overline{\text{IRQ2}}$	External interrupt request	0x000C
ICIA	Input capture A (16 bit timer)	0x0018
ICIB	Input capture B (16 bit timer)	0x001A
ICIC	Input capture C (16 bit timer)	0x001C
ICID	Input capture D (16 bit timer)	0x001E
OCIA	Output compare A (16 bit timer)	0x0020
OCIB	Output compare B (16 bit timer)	0x0022
FOVI	Overflow (16 bit timer)	0x0024
CMI0A	Compare-match A (8 bit timer0)	0x0026
CMI0B	Compare-match B (8 bit timer0)	0x0028
OVI0	Overflow (8 bit timer0)	0x002A
CMI1A	Compare-match A (8 bit timer1)	0x002C
CMI1B	Compare-match B (8 bit timer1)	0x002E
OVI1	Overflow (8 bit timer1)	0x0030
ERI	Receive error	0x0036
RXI	Receive end	0x0038
TXI	TDR empty	0x003A
TEI	TSR empty	0x003C
A/D	A/D conversion end	0x0046
WOVF	Watchdog timer overflow	0x0048

I/O Ports:

- 43 input/output lines (16 of which can drive LEDs)
- 8 input-only lines

Power-down modes:

- Sleep mode
- Software standby mode
- Hardware standby mode

12 RCX Hardware Portrait

CPU: All the H8 CPU functions are accessible.

Memory: The RCX has 32k external RAM. It configures memory operating mode 2 with the following address room:

- 0x0000 - 0x3FFF : 16k On-chip ROM (vector and data tables, RCX-executive and basic subroutines)
- 0x4000 - 0x7FFF : reserved; may not be accessed; addresses do not physically exist with the H8/3292
- 0x8000 - 0xCBFF : Firmware & user code area: 19k external address space RAM
- 0xCC00 - 0xEE5D : User data: 8797bytes in external RAM
- 0xEE5E - 0xEFFF : external RAM used by on-chip ROM functions
- 0xF000 : motor control byte; bits 7,6 are related to motor A; bits 3,2 to motor B; bits 1,0 to motor C
- 0xF001 - 0xFB7F : unusable external RAM; writing to this space affects the motors
- 0xFB80 - 0xFD7F : reserved; may not be accessed; addresses do not physically exist with the H8/3292
- 0xFD80 - 0xFDBF : on-chip 64 bytes RAM used by on-chip ROM functions (shadow-registers, vectors, data)
- 0xFDC0 - 0xFF7F : on-chip 448 bytes RAM used as stack (Note that the access to this RAM is much faster than to the external RAM. So, it can be used as cache-memory.)
- 0xFF80 - 0xFF87 : unusable external RAM; writing to this space affects the motors
- 0xFF88 - 0xFFFF : on-chip register field used to configure the H8-devices

16-bit free-running timer (1 channel): The 16-bit free-running timer is configured to generate an interrupt each millisecond and execute several input or output (I/O) routines.

8-bit timer (2 channels): Both timers control their related outputs on hardware level without interrupt.

- 8-bit Timer0 is used to produce sound with the RCX speaker.
- Timer1 generates the 38,5kHz carrier necessary for the infrared communication.

Watchdog timer (1 channel): By default, the WDT is not configured with the RCX. However, it is well and truly accessible.

Serial communication interface (SCI) (1 channel): The SCI-module is configured in asynchronous mode at 2400baud, 8bit, 1 stop-bit, odd parity. The device is used in half-duplex mode managed by software means. (Since the infrared communication road represents a single channel for bi-directional data exchange, the software must take care that the RCX does not transmit while receiving.)

A/D converter (ADC) (8 channels): The RCX uses 4 ADC channels: sensor 3 (channel 0), sensor 2 (channel 1), sensor 1 (channel 2), battery level (channel 3).

Interrupts: The RCX uses (or may use) the following selection of interrupts, all of which may be redefined:

Name	RCX vector address	Interrupt Service Routine address	Description
$\overline{\mathrm{NMI}}$	0xFD92	not implemented	Non maskable interrupt
$\overline{\mathrm{IRQ0}}$	0xFD94	0x1AB8	Run button handler
$\overline{\mathrm{IRQ1}}$	0xFD96	0x294A	On/Off button handler
OCIA	0xFDA2	0x36BA	16-bit Timer Output Compare A handler (basic RCX clock)
OCIB	0xFDA4	not implemented	16-bit Timer Output Compare B interrupt (not used)
FOVI	0xFDA6	not implemented	16-bit Timer Overflow interrupt (not used)
CMI0A	0xFDA8	not implemented	8-bit Timer 0 Compare Match A interrupt (not used)
CMI0B	0xFDAA	not implemented	8-bit Timer 0 Compare Match B interrupt (not used)
OVI0	0xFDAC	not implemented	8-bit Timer 0 Overflow interrupt (not used)
CMI1A	0xFDAE	not implemented	8-bit Timer 1 Compare Match A interrupt (not used)
CMI1B	0xFDB0	not implemented	8-bit Timer 1 Compare Match B interrupt (not used)
OVI1	0xFDB2	not implemented	8-bit Timer 1 Overflow interrupt (not used)
ERI	0xFDB4	0x30A4	Serial Receive Error handler
RXI	0xFDB6	0x2C10	Serial Receive End handler
TXI	0xFDB8	0x2A9C	Serial Ready to Transmit handler
TEI	0xFDBA	0x2A84	Serial Transmit Error handler
ADI	0xFDBC	0x3B74	A/D Conversion End handler
WOVF	0xFDBE	not implemented	Interval Timer Overflow interrupt (Watchdog-timer)

RCX Port definitions:

Port	Bit	I/O	Description
1	7..0	O	Address bus
2	7..0	O	Address bus
3	7..0	I/O	Data bus
4	0	O	Transmitter range (0 = long; 1= short)
4	1	I	On/off button ($\overline{\mathrm{IRQ1}}$; 0 = pressed)
4	2	I	Run button input ($\overline{\mathrm{IRQ0}}$; 0 =pressed)
4	3	O	Bus read ($\overline{\mathrm{RD}}$; 0 = CPU is reading at an externaladdress.)
4	4	O	Bus write ($\overline{\mathrm{WR}}$; 0 = CPU is writing to an external address.)
4	5	O	Bus address strobe ($\overline{\mathrm{AS}}$; 0 = there is a valid address on the address bus)
4	6	O	System clock for external devices
4	7	I	Bus wait ($\overline{\mathrm{WAIT}}$; 0 =requests the CPU to insert wait states into the bus cycle while accessing an external address)
5	0	O	Transmit data (TxD)
5	1	I	Receive data (RxD)
5	2	O	External device power control (0 = power on ; RAM, sensor pull-ups...)
6	0	O	Sensor 3 9V power (0 = power off)
6	1	O	Sensor 2 9V power (0 = power off)
6	2	O	Sensor 1 9V power (0 = power off)
6	3	O	Unused (must be configured to output!)
6	4	O	Speaker (TMO0) (0 = speaker LOW = -5V; 1 = High = +5V)
6	5	I/O	LCD input/output
6	6	I/O	LCD input/output
6	7	O	Infrared carrier (TMO1) (0 = infrared LEDs off; 1 = LEDs on)
7	0	I	Sensor 3 input (AN0)
7	1	I	Sensor 2 input (AN1)
7	2	I	Sensor 1 input (AN2)
7	3	I	Battery voltage input (AN3)
7	4	I	Unused
7	5	I	Unused
7	6	I	View button input (0 = pressed)
7	7	I	Prgm button input (0 = pressed)

Power-down modes: The RCX is powered down in "Software standby mode". The system clock stops and chip functions halt, including the CPU and the on-chip supporting modules. Power consumption is drastically reduced. The on-chip modules and their registers are reset to their initial states. However the contents of the CPU registers and on-chip RAM remain unchanged as long as minimum necessary voltage supply is maintained. This means that the CPU continues in software from the point, where the controller was sent to standby. The RCX can be brought out of software standby mode through external interrupt requests IRQ0 (Run-button) or IRQ1 (On/Off-button). On the RCX hardware level it is therefore necessary that at least the Run-button is always powered. In fact all four buttons are permanently connected to the internally stabilized 5V-supply.

13 Dual task firmware

The graphical code shown in Fig. 11 obeys the ROBOLAB graphical syntax.[11] In fact, it is a dialect called Ultimate Robolab.[12] The green traffic light always marks the beginning of the RCX program. Hidden from the user's eyes, a bunch of initialization processes are added to the code. Ultimate Robolab also creates a special system task running in the background that should handle interrupt-driven RCX states. For instance, if a new valid sensor reading is available from the H8/3292 ADC module, this background process will refill the related memory locations with the new values. The current handler only manages sensor readings and display update, among which the tiny running man in the right part of the display (cf. Fig 9). The function of this figure is to show that the firmware is still correctly running, and didn't get hooked somewhere in the code. After the green light icon follows sensor 1 configuration as a light sensor, whose values are sent to the RCX display buffer. The display function is placed between the jump and land icons, indicating that the RCX should repeat this process over and over again. The red traffic light notifies the end of the program, initiating code generation, compilation and download.

In fact, this firmware runs the main task and the background handler in parallel, representing a much higher level of complexity than the previous examples. The resulting code is impressively long, because of the hidden functionalities.

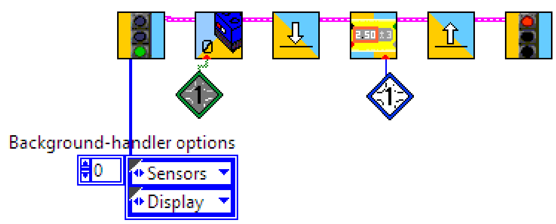


Figure 11: Ultimate Robolab easy-to-understand graphical code.

test_light_sensor.srec, part I (S-record file generated by Ultimate Robolab from the graphical code shown in Fig. 11).

```
S01300003F4C49425F56455253494F4E5F4C303046
S1138000790101F41B01790202941B020D2246FA44
S1138010 0D1146F0790600006B86FD8E7900CC4088
S11380207901F0005E0004365E0091AA5E0091586A
S11380307907FF7E5E00885204806A08FF90E8F7A3
S11380406A88FF9079068F226B86FD94067F7901FA
S113805000641B01790202941B020D2246FA0D11E1
S113806046F05E0027AC5E0027C8790630065E0045
S11380701B625E0027C8790600006B86CD2AFE00CD
S11380806A8ECC86FE566A8ECC87FE776A8ECC8842
S1138090FE676A8ECC89FE646A8ECC8A5E00935039
S11380A05E0027C879008FE8790600006986 7900A8
S11380B08FC67906000069866A0EFFC7CE016A8EF4
S11380C0FFC76B05FD8E6A0ECD2C1CDE47045A00DB
S11380D080DA5E008F7E5A0080C2790600015E005D
S11380E09628FEFF6A8ECD2C790600006B86CD2A79
S11380F0FE006A8ECC86FE006A8ECC87FE006A8EF5
S1138100CC88FE006A8ECC89FE006A8ECC8A5E0023
S113811093505E0027C87906EE2E6B86CD40FE0095
S11381206B00CD40688E7906EF936B86CD4279065E
S1138130EFB66B86CD447906EE686B86EFB8FE002A
S11381406B00EFB8688E7906CD306B86CD46790625
S1138150EE326B86CD487906EE386B86CD4A7906CA
S1138160EE3E6B86CD4C7906EE336B86CD4E7906AB
S1138170EE396B86CD507906EE3F6B86CD52FEFF0E
```

```
S11381806A8ECD54FEFF6A8ECD555E0094AC04809A
S113819079060 0016B86CD6AFE006B00CD6A79011A
S11381A0CD0C0910688E067F04806A08FF90C8080A
S11381B06A88FF90FE006A8ECD6C790623286B8651
S11381C0CD6E79050000790600006B85CD706B8656
S11381D0CD72FE006A8ECD74790600006B86CD7673
S11381E0FE006A8ECD78FE006A8ECD797900CD309F
S11381F0790200007901CD4069820B80190146F4B0
S1138200790600006B86CD7A790600006B86CD7CFC
S1138210FE006A8ECD7E790600006B86CD80FE0060
S11382206A8ECD82FE196A8ECD83FE006A8ECD845F
S1138230067F790100321B01790202941B020D2292
S113824046FA0D1146F0790500806B06EE641D5664
S11382504404 5A0083127900000A19076FF70008D4
S11382607905EE44FC070D767B5C598F6F7600082A
S1138270 6DF6790610005E0014C00B870D75FC07C1
S11382807906EE447B5C598F7900000A0907790070
S1138290000A19076FF700087905EE4CFC070D7606
S11382A07B5C598F6F7600086DF6790610015E00CF
S11382B014C00B870D75FC077906EE4C7B5C598F59
S11382C07900000A09077900000A19076FF7000808
S11382D07905EE54FC070D767B5C598F6F760008AA
S11382E06DF6790610025E0014C00B870D75FC074F
S11382F07906EE547B5C598F7900000A09077905EB
S1138300007F6B06EE6416DE16566B86EE645A002D
```

```
S113831083127905000A6B06EFD21D5644045A00F8
S113832087A6FD016A0ECD7E08DE6A8ECD7EFD0A2E
S11383306A0ECD7E1CDE44045A00879AFD016A0E46
S1138340CD8508DE6A8ECD856A0DCD836A0ECD8519
S11383501CDE44045A008466FE006A8ECD857906CF
S1138360301A5E001E4A790500006B06CD801D564D
S113837046045A0084406B06CD865E0097F06B85FB
S1138380CD886B86CD8A79030000790400046B05E2
S1138390CD886B06CD8A5E0001FE6B85CD886B862C
S11383A0CD8A6B06CD805E0097F00D530D646B0591
S11383B0CD886B06CD8A5E0003066B85CD886B8602
S11383C0CD8A79030000790400036B05CD886B0623
S11383D0CD8A19461EBD1E354310790500007906 68
S11383E000036B85CD886B86CD8A79030000790403
S11383F000016B05CD886B06CD8A09460EBD0E3591
S11384006B85CD886B86CD8A790630185E001E4A52
S1138410FE006A8ECD8C6B03CD886B05CD8A6A0E0B
S1138420CD8C1CDE45045A00843C0A0E6A8ECD8C2D
S1138430790630185E001B625A0084165A00846266
S1138440FD006A0ECD821CDE47045A00845A79066C
S113845030185E001E4A5A008462790630185E00A9
S11384601B625A008466FD006A0ECD741CDE470450
S11384705A0084D0790630065E001B62FD136B0043
S1138480CD42680E1CDE47045A0084CCFD016A0E02
S1138490CD8D1CDE47045A0084A67906301D5E008F
```

[11] http://www.legoengineering.com/platform/robolab/, [retrieved 11/2022].
[12] https://www.convict.lu/Jeunes/ultimate_stuff/Ultimate_intro.htm, [retrieved 11/2022].

test_light_sensor.srec, part II

```
S11384A01B625A0084AE7906301C5E001B62FDA47C
S11384B06A0ECD791CDE47045A0084C87906301A4A
S11384C05E001B625A0084C85A0084CC5A0084E8BB
S11384D0790630075E001B627906301C5E001E4A7A
S11384E07906301D5E001E4A6A0ECD74170E6A8E24
S11384F0CD7479050000 6B06CD2A1D5646045A003E
S11385008570FD016A0ECD8E1CDE47045A0085443E
S11385106A0ECD8F5E0097F65E0097FA79053002FE
S113852009566DF66B00CD2A680E5E0097F65E0069
S113853097FA6DF6790630015E001FF26D760B87B4
S11385405A00856C6A0ECD8F5E0097F65E0097FA33
S11385507905300209566DF66B00CD2A69066DF676
S1138560790630015E001FF26D760B875A00857029
S1138570FD006A0ECD541CDE4C045A008768FD00D6
S11385806A0ECD541CDE46045A0086D4FD016A0EE5
S1138590CD541CDE46045A008640FD026A0ECD54BF
S11385A01CDE46045A0085AC5A00863CFD016B0078
S11385B0CD52680E1CDE46045A00862CFD026B006D
S11385C0CD52680E1CDE46045A008618FD036B0070
S11385D0CD52680E1CDE46045A008604FD046B0073
S11385E0CD52680E1CDE46045A0085F05A00860004
S11385F0790630165E001E4A790630155E001E4A67
S11386005A008614790630165E001B627906301514
S11386105E001B625A008628790630155E001B62DA
S1138620790630165E001E4A5A00863C79063016E0
S11386305E001B62790630155E001E4A5A0086D027
S1138640FD016B00CD50680E1CDE46045A0086C04C
S1138650FD026B00CD50680E1CDE46045A0086AC4F
S1138660FD036B00CD50680E1CDE46045A00869852
S1138670FD046B00CD50680E1CDE46045A00868455
S11386805A008694790630135E001E4A790630122F
S11386905E001E4A5A0086A8790630135E001B62F1
S11386A0790630125E001B625A0086BC79063012D3
S11386B05E001B62790630135E001E4A5A0086D0A9
S11386C0790630135E001B62790630125E001E4A88
S11386D05A008764FD016B00CD4E680E1CDE460419
S11386E05A008754FD026B00CD4E680E1CDE460418
S11386F05A008740FD036B00CD4E680E1CDE46041B
S11387005A00872CFD046B00CD4E680E1CDE46041E
S11387105A0087185A008728790630105E001E4AD5
S11387207906300F5E001E4A5A00873C79063010EC
S11387305E001B627906300F5E001B625A00875097
S11387407906300F5E001B62790630105E001E4A0E
S11387505A008764790630105E001B627906300F7F
S11387605E001E4A5A0087680480FE016B00CD90B2
S11387707901CD0C0910688E067F5E0027C8048044
S1138780FE006B00CD907901CD0C0910688E067F3F
S1138790FE006A8ECD7E5A00879A790600006B86B0
S11387A0EFD25A0087A65A0082465A0087AA048053
S11387B07906EE456B86CD92FE806B00CD92688E7C
S11387C07906EE486B86CD92067F0480790610000F
S11387D05E0019467906EE446B86CD92FE036B0072
S11387E0CD92688E7906EE486B86CD92067FFE02AD
S11387F06A8ECD8E7906EE486B86CD2AFE006A8E96
S1138800CD8F5A0087EE0480790500016B06CD966A
S113881019566B86CD965E0094CE048079060001D5
S11388206B86CD6AFE026B00CD6A7901CD0C091016
S11388300688E067FFD026B00CD907901CD0C09108E
S1138840680E1CDE47045A00884E5A0088345A00D1
S1138850884E790600006B86CC8C790600016B860D
S1138860CC8E790600016B86CC90790600016B8674
S1138870CC92790600206B86CC94790604016B8639
S1138880CC96790601016B86CC98790601016B8642
S1138890CC9A790602046B86CC9C790610416B86D7
S11388A0CC9E790604106B86CCA0790684216B865D
S11388B0CCA2790608446B86CCA4790622116B867F
S11388C0CCA6790611116B86CCA8790611116B86A2
S11388D0CCAA790611226B86CCAC790644896B86CE
S11388E0CCAE790624916B86CCB0790624896B8654
S11388F0CCB2790624926B86CCB4790652496B864D
S1138900CCB6790624A56B86CCB8790625256B8673
S1138910CCBA790624A96B86CCBC79064A956B86C2
S1138920CCBE79062A546B86CCC07906AA556B86D9
S1138930CCC27906554A6B86CCC47906A9556B86A1
S1138940CCC6790655556B86CCC8790655556B86D2
S1138950CCCA7906AAB56B86CCCC790656AA6B86AF
S1138960CCCE7906D5AB6B86CCD0790655AA6B8677
S1138970CCD27906DB566B86CCD47906B56A6B868E
S1138980CCD67906DB5A6B86CCD87906DADA6B86DD
S1138990CCDA7906DB6D6B86CCDC7906ADB66B8603
S11389A0CCDE7906DB6E6B86CCE07906DB766B86FC
S11389B0CCE27906EEDD6B86CCE47906BB766B8682
S11389C0CCE67906EEEE6B86CCE87906EEEE6B86AE
S11389D0CCEA7906F7BB6B86CCEC7906DDEE6B86D1
S11389E0CCEE7906FBEF6B86CCF079067BDE6B86F3
S11389F0CCF27906FDFB6B86CCF47906EFBE6B8679
S1138A00CCF67906FEFE6B86CCF87906FEFE6B860E
S1138A10CCFA7906FFDF6B86CCFC7906FBFE6B8617
S1138A20CCFE7906FFFE6B86CD007906FFFE6B86DB
S1138A30CD027906FFFF6B86CD047906FFFE6B86C1
S1138A40CD067906FFFF6B86CD087906FFFF6B86A8
S1138A50CD0A790100C81B01790202941B020D228A
S1138A6046FA0D1146F004807906EE646DF6790641
S1138A70EE745E003B9A0B875E00149804807906C8
S1138A8010005E0019C47906EE446B86CD92FE00A2
S1138A906B00CD92688E7906EE486B86CD92067F92
S1138AA004807906EE456B86CD92FE006B00CD927E
S1138AB0688E7906EE486B86CD92067F0480790639
S1138AC0EE486B86CD92790600006B00CD926986EE
S1138AD0067F0480790610015E0019C47906EE4C0F
S1138AE06B86CD98FE006B00CD98688E7906EE50B5
S1138AF06B86CD98067F04807906EE4D6B86CD980D
S1138B00FE006B00CD98688E7906EE506B86CD9895
S1138B10067F04807906EE506B86CD9879060000C1
S1138B206B00CD986986067F0480790610025E0095
S1138B3019C47906EE546B86CD9AFE006B00CD9A76
S1138B40688E7906EE586B86CD9A067F0480790691
S1138B50EE556B86CD9AFE006B00CD9A688E79063C
S1138B60EE586B86CD9A067F04807906EE586B86AF
S1138B70CD9A790600006B00CD9A6986067FFEFFD3
S1138B80790570015E009750FEFF790570025E006D
S1138B909750FEFF790570035E009750790600043F
S1138BA0790570015E0096A27906000479057002D4
S1138BB05E0096A279060004790570035E0096A21C
S1138BC05E001ABA5E002964790600016DF679062D
S1138BD000016DF67906EE746DF67906EE645E00C5
S1138BE030D00B870B870B87790617705E003266DA
S1138BF0FE006A8ECD8D5E0036920480FE036A8E89
S1138C00EE5BFE1E6A8EEE53FE086A8EFFC8FE0308
S1138C106A8EFFC96A0EFFC3EEFE6A8EFFC379063D
S1138C208D0C6B86FDA27906FF946B86CD9E790636
S1138C3001F46B00CD9E69867906FF926B86CD9E16
S1138C40790600006B00CD9E69860480FE006A8E6E
S1138C50CD0CFE026A8ECD0D7906FF7E6B86CDA017
S1138C607906FE9E6B86CDA2790600026B86CDA4AE
S1138C70790600006B86CD906B00CD90100812003D
S1138C807901CDA009106987790600006B86CD902F
S1138C90790500016B06CD901D5645045A008CF0FD
S1138CA00B066B86CD906B00CD9010081200790101
S1138CB0CDA0091069076B00CD901008120079015A
S1138CC0CD16091069066DF6790600006DF66DF699
S1138CD0790604D46DF67901000C19176B00CD9064
S1138CE0100812007901CDA0091069875A008C90FC
S1138CF0790600006B86CD906B00CD9010081200BD
S1138D007901CDA009106907067F54706DF06DF1F8
S1138D106DF26DF36DF46DF56A0EFF91EEF76A8EF5
S1138D20FF916A0EEFCFF6006A0DEE5B51D60C6E2F
S1138D30460A6A0EFFBBEEF86A8EFFBB6A0EEE5C60
S1138D406A8EF000FE006A8EEE5C6B05EE346B0601
S1138D50EE36770E5E00980444106A0AEFCAEAC04E
S1138D606A02EE5C142A6A8AEE5C6B85EE346B86D7
S1138D70EE366B05EE3A6B06EE3C770E5E00980426
S1138D8044106A0AEFCAEA0C6A02EE5C142A6A8A8D
S1138D90EE5C6B85EE3A6B86EE3C6B05EE406B0650
S1138DA0EE42770E5E00980444106A0AEFCAEA03AF
S1138DB06A02EE5C142A6A8AEE5C6B85EE406B867B
S1138DC0EE426A0EEFCFF6006A0DEE5B51D60C6EEF
S1138DD0460A6A0EFFE8CE606A8EFFE86A0EEFCFAA
S1138DE00A0E6A8EEFCF6A06EE531C6E4306FE013B
S1138DF06A8EEFCF6B06EFD20B066B86EFD26A0E59
S1138E00EFD00A0E6A8EEFD0AE0A452EFE006A8EBD
S1138E10EFD06B06EE760B066B86EE766B06EE788B
S1138E200B066B86EE786B06EE7A0B066B86EE7AA1
S1138E306B06EE7C0B066B86EE7CFD016B00CD902F
S1138E407901CD0C0910680E1CDE46045A008F1409
S1138E506B00CD90100812007901CDA0091069873A
S1138E60FD006A0ECDA61CDE47045A008E826B0604
S1138E70CD906B86CDA8790600006B86CD905A0012
S1138E808EE46B06CDA86B86CD90FE006A8ECDAAD9
S1138E90FD016A0ECDAA08DE6A8ECDAA790500011B
S1138EA06B06CD9009566B86CD90790500026B0660
S1138EB0CD901D564508790600016B86CD90FDFFD5
S1138EC06A0ECDAA1CDE450604805E008F22FD02E6
S1138ED06B00CD907901CD0C0910680E1CDE4604AE
S1138EE05A008E90FD016A0ECDA608DE6A8ECDA6DA
S1138EF0FD016A0ECDA61CDE4306FE006A8ECDA6E7
S1138F006B00CD90100812007901CDA0091069070A
S1138F105A008F146D756D746D736D726D716D7022
S1138F2054706DF06DF16DF26DF36DF46DF56A0ED3
S1138F30CD2C170E6A8ECD2CFD006A0ECD2C1CDEC5
S1138F40462E04805E0036AA5E0036365E0027F4B3
S1138F505E001A225E003ED4790100FA1B01790207
S1138F6002941B020D2246FA0D1146F05A00801CA0
S1138F706D756D746D736D726D716D70547079067C
S1138F80EE306DF6790640005E0029F20B876B0630
S1138F90EE306B86EE2C7906EE2A6DF6790630000A
S1138FA05E001FB60B876B06EE2C110EDE01100E60
S1138FB0100E100E6B04EE2A14CE6B86EE2C6B069B
S1138FC0EE2C6B86EE280000790500006B06EE307E
S1138FD01D5647045A0091567905000C6B06EE288C
S1138FE01D5647045A0090300000790100781B01A6
S1138FF0790202941B020D2246FA0D1146F0048007
S11390005E0036AA5E0036365E0027F45E001A2251
S11390105E003ED40480790101F41B0179020294CC
S11390201B020D2246FA0D1146F05F005A009152D0
S11390300480FE016B00CD907901CD0C0910688E8F
S1139040067F5E0027AC5E0027C80480FE006B003C
S1139050CD907901CD0C0910688E067F790100322C
S11390601B01790202941B020D2246FA0D1146F0FF
S113907004805E0036AA5E0036365E0027F45E0099
S11390801A225E003ED4790100781B017902029421
S11390901B020D2246FA0D1146F07907FF7E790680
S11390A036BA6B86FDA25E002964790100641B0167
S11390B0790202941B020D2246FA0D1146F0067F46
S11390C05E002A62790600016B86EE30067F067F29
S11390D0067F067F067F067F067F067F067F79067A
S11390E000016B86EE307906EE646DF67906EE7467
S11390F05E003B9A0B877906000A6DF6790607D075
S11391006DF6790617735E00327C0B870B87790156
S1139110006E1B01790202941B020D2246FA0D1117
S113912046F07906000A6DF6790605466DF679067E
S113913017735E00327C0B870B877901006E1B017E
S1139140790202941B020D2246FA0D1146F05A00E1
S1139150801C5A00915654707906801C6B86CD168C
S1139160790687AE6B86CD18790600006B86CD1A2B
S1139170790600006B86CD1C790600006B86CD1E48
S1139180790600006B86CD20790600006B86CD2230
S1139190790600006B86CD24790600006B86CD2618
S11391A0790600006B86CD285470FE016A8ECC40A0
S11391B0FE006A8ECC41FE016A8ECC42FE036A8EBB
S11391C0CC43FE036A8ECC44FE016A8ECC45FE008E
S11391D06A8ECC46FE016A8ECC47FE036A8ECC487B
S11391E0FE036A8ECC49FE016A8ECC4AFE006A8E7B
S11391F0CC4BFE016A8ECC4CFE036A8ECC4DFE0343
S11392006A8ECC4EFE026A8ECC4FFE046A8ECC5031
S1139210FE056A8ECC51FE086A8ECC52FE076A8E2B
S1139220CC53FE026A8ECC54FE046A8ECC55FE05F7
S11392306A8ECC56FE086A8ECC57FE076A8ECC58E0
S1139240FE046A8ECC59FE056A8ECC5AFE066A8EF0
S1139250CC5BFE086A8ECC5CFE076A8ECC5DFE04A7
S11392606A8ECC5EFE056A8ECC5FFE066A8ECC609C
S1139270FE086A8ECC61FE076A8ECC62FE106A8EA0
S1139280CC63FE106A8ECC64FE016A8ECC65FE0160
S11392906A8ECC66FE106A8ECC67FE206A8ECC682F
S11392A0FE206A8ECC69FE026A8ECC6AFE026A8E5B
S11392B0CC6BFE206A8ECC6CFE806A8ECC6DFE800A
S11392C06A8ECC6EFE086A8ECC6FFE086A8ECC7007
S11392D0FE806A8ECC71FE206A8ECC72FE026A8E9D
S11392E0CC73FE026A8ECC74FE026A8ECC75FE02DC
S11392F06A8ECC76FE806A8ECC77FE086A8ECC7847
S1139300FE086A8ECC79FE086A8ECC7AFE086A8EE7
S1139310CC7BFE206A8ECC7CFE206A8ECC7DFE203A
S11393206A8ECC7EFE206A8ECC7FFE206A8ECC8047
S1139330FE806A8ECC81FE806A8ECC82FE806A8E3F
S1139340CC83FE806A8ECC84FE806A8ECC8554708C
S1139350790600056B86CDAC790500006B06CDACC6
S11393601D5642045A0094AA1B066B86CDACFE40F2
S11393706A8ECDAE790600076B86CDB07905000017
S11393806B06CDB01D5642045A0094A61B066B869F
S1139390CDB06B00CDAC7901CC860910680E6A8E28
S11393A0CDB26A0DCDAE6A0ECDB216DE6A8ECDB2F9
S11393B06B06CDB06B86CDB4790500056B06CDB4E7
S11393C05E0001306B86CDB47905CC406B06CDB42F
S11393D009566B86CDB46B06CDAC6B86CDB66B05FD
S11393E0CDB46B06CDB609566B86CDB66B00CDB656
S11393F0680E5E0097F66B86CDB87905EF436B0684
S1139400CDB809566B86CDB86B06CDB06B86CDB4B2
S1139410790500056B06CDB45E0001306B86CDB4E6
S11394207905CC636B06CDB409566B86CDB46B066B
S1139430CDAC6B86CDB66B05CDB46B06CDB609560B
S11394406B86CDB66B00CDB6680E6A8ECDBAFD00D8
S11394506A0ECDB21CDE47045A0094866A0ECDBA6D
S1139460170E6A8ECDBA6A0DCDBA6B00CDB8680E04
S113947016DE6B00CDB8688E6A0ECDBA170E6A8E06
S1139480CDBA5A0094986A0DCDBA6B00CDB8680E7B
S113949014DE6B00CDB8688E6A0ECDAE110E6A8EFA
S11394A0CDAE5A00937C5A0093585470790600520E
S11394B06B86CDBC790600526B86CDBE7906000175
S11394C06B86CD96790600016B86CDC054706B0625
S11394D0CDBE6B86CDC26B05CDC06B06CDC25E0036
S11394E001306B86CDC27905000A6B06CDC25E00F5
S11394F001306B86CDC26B05CD966B06CDC25E009A
S113950001BE6B86CDC2790500056B06CDC209564B
S11395106B86CDC27905000A6B06CDC25E0001BE37
S11395206B86CDC279057FFF6B06CDC21D56430812
S113953079067FFF6B86CDC2790500016B06CDC240
S11395401D564408790600016B86CDC26B06CDC26D
S11395506B86CDBE6B06CD966B86CDC05470790011
S1139560000C19076FF500006FF600026B06EE7C3A
S11395705E0097F05E00980E6FF500046FF6000640
S113958079050000790600006FF500086FF6000A14
S11395906F7300006F7400026F7500086F76000A3A
S11395A019461EBD1E354F045A0096206B06EE7C01
```

test_light_sensor.srec, part III


```
S11395B05E0097F05E00980E6FF500086FF6000AF8
S11395C06F7300046F7400066F7500086F76000A02
S11395D019461EBD1E356FF500086FF6000A7903B8
S11395E00000790400006F7500086F76000A1946D5
S11395F01EBD1E354D045A00961C790300007904F8
S1139600FFFF6F7500086F76000A09460EBD0E3536
S11396106FF500086FF6000A5A00961C5A009590F6
S11396207900000C0907547079000081 9076FF6ED
S113963000007905000006B06CD901D5647045A00D8
S1139640964E6B06CDBC6FF600025A0096566B0630
S1139650CDBE6FF60002790600006FF600046F755E
S113966000006F7600041D5645045A00969A0B06CC
S11396706FF60004790600006FF600066F750002C3
S11396806F7600061D5645045A0096960B066FF649
S113969000065A00967C5A00965E7900000809078B
S11396A0547004806B85CDC46B86CDC66B06CDC47D
S11396B06B86CDC8790500036B06CDC816DE16564F
S11396C06B86CDC8790500066B06CDC85E0001300D
S11396D06B86CDC87905EE2C6B06CDC809566B8628
S11396E0CDC86B00CDC86E0E00015E0097F66B869E
S11396F0CDCA6B05CDC66B06CDCA09566B86CDCAF3
S11397007905000036B06CDCA1D5646246B00CDC806
S11397106E0E00005E0097F66B86CDCC6B06CDCC61
S113972011061 30E6B86CDCC6B06CDCC5E00962864
S11397306B06CDC66B00CDC86E8E00016B06CDC835
S11397406DF66B06CDC45E003CE60B87067F54706C
S11397500 4806B85CDC46A8ECDCE6B06CDC46B8691
S1139760CDC8790500036B06CDC816DE16566B869F
S1139770CDC8790500066B06CDC85E0001306B865D
S1139780CDC87905EE2C6B06CDC809566B86CDC8D4
S11397906A0DCDCE6B00CDC8680E1CDE474E6A0E4D
S11397A0CDCE6B00CDC8688E6B00CDC8680E5E0067
S11397B097F66B86CDD0790500086B06CDD05E00AF
S11397C001BE6B86CDD06B00CDD01008120010081 5
S11397D012007901CC8C091069056F0600026B004F
S11397E0CDC86F8500020B806F860002067F547036
S11397F0790500005470F600547073764704790 6CD
S11398007FFF54701305130D1306130E547073750C
S113981047087905 7FFF7906FFFF54700000446F1D
S11398202079 6F7520627974652C207768656E20DD
S113983049206B6E6F636B3F0000000000000000007E
S90380007C
```


14 The World's unique RCX Virus

Back to 2005, a discussion came up in the LEGO robotics community, whether robots of the future might become victims of malware infections, just like their badly suffering cousins, the general-purpose computers. The author of the present paper was intrigued by this question and started playing with the idea of providing a practical proof that, depending on their architecture, embedded systems like the RCX might well be captured by self-replicating programs.

A necessary condition for program self-replication –besides sufficient memory space– is the possibility of automated exchange of data and instruction code, as the self-replicating program must view its own code as data that it can manipulate. Computer systems that are based on von Neumann architecture fulfill this condition, because -roughly said- data, addresses and program instructions share the same memory space and are fetched and stored in the same manner via the same data bus. Note the difference to the Harvard architecture, which fosters the separation of instructions and data in memory and their transportation pathways.[13]

The H8/3292 CPU follows the von Neumann concept. The author therefore supposed that a series of RCXs in IR-proximity would create a data space that is big enough to allow program self-replication. The reason, he thought, was that during the boot-loading process, firmware code is transported and stored as if it were simple data. Hence, with the adequate code, the program could send a copy of its own instruction sequence from the firmware memory-space to the neighbor RCXs. The such-wise infected RCXs would store the data as a new firmware, and –once safely loaded and unlocked– run it. By this manner the code would jump from RCX to RCX on a virtual daisy chain.

The practical proof of self-replication would be made by writing such a firmware. The software should present the following features:

- No physical damage what so ever should be made, except for the fact that the virus firmware would erase all user data and spread itself without any further human help.
- The virus firmware (vfw) should write the text "LEGO" to the display, so that the user thinks, he or she got a normal (ULTIMATE ROBOLAB) firmware.
- The vfw should wait a few seconds, then mirror the text by writing "O@3J", which on the 7-digit display represents the mirrored mystical word "LEGO". This should produce a *Shocking*-effect.
- Now the vfw should wait for a further random duration while checking, if another RCX is occupied with sending. This should make sure that never more than one RCX is sending at the same time.

[13]Interestingly, researchers have proven that embedded system based on hybrid Harvard architectures could be successfully infected by malware. Although devices with strict separation of instruction code and data memory are well immunized, for practical reasons, most modern Harvard-based microcontrollers nowadays allow changes of the instructions through data memory. (cf. for instance: A. Francillon, *Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks*, Doctoral Thesis, Networking and Internet Architecture [cs.NI], Institut National Polytechnique de Grenoble - INPG, (2009), https://tel.archives-ouvertes.fr/tel-00540371/document, retrieved [11/2022].

- If the channel is free, the vfw should send a few *Ping* messages (opcode 0x10). Neighbor RCXs running either the ROM executive or standard / third party enhanced firmware would reply altogether to this message.
- The vfw should ignore any RCX reply.
- The vfw should also send few broadcasting messages (opcode 0xF7) with parameter '1'. (Normally RCXs don't reply on this message. The non-zero value should tell the vfw-infected colleague that another RCX is sending out the vfw.
- The vfw should in fact behave as a self-replicating firmware downloader program:
  - Start the firmware download protocol, while choosing the address room of the vfw itself.
  - Replies from receiving RCXs should be ignored.
  - RCXs, which are either running the ROM executive or the standard firmware should by this means go into boot mode.
  - The vfw code should be cut into slices of 200 bytes, which are wrapped into valid RCX IR-packets and are sent out.
  - The display should show the current byte number.
  - At the end unlock the remote vfw and restart the process.
- Internally the vfw would set the message value to '2', just for the display task.
- A separate task should update the display with the current broadcast message. (This would be needed during the debugging phase.)
- The background handler should continuously verify (and handle it), if a valid broadcasting message 0xF7 has been received.
- Buttons should be disabled, so that the user cannot stop the process.

Although the author is aware that it will be difficult for most readers without ROBOLAB experience to fully understand the code, he wanted to present the complete graphical code here. From the feature list follows that 3 independent tasks are running in this firmware, background handler included. Visibly, the opcode 0x45 wrapping certainly is the most complicated part of the program.

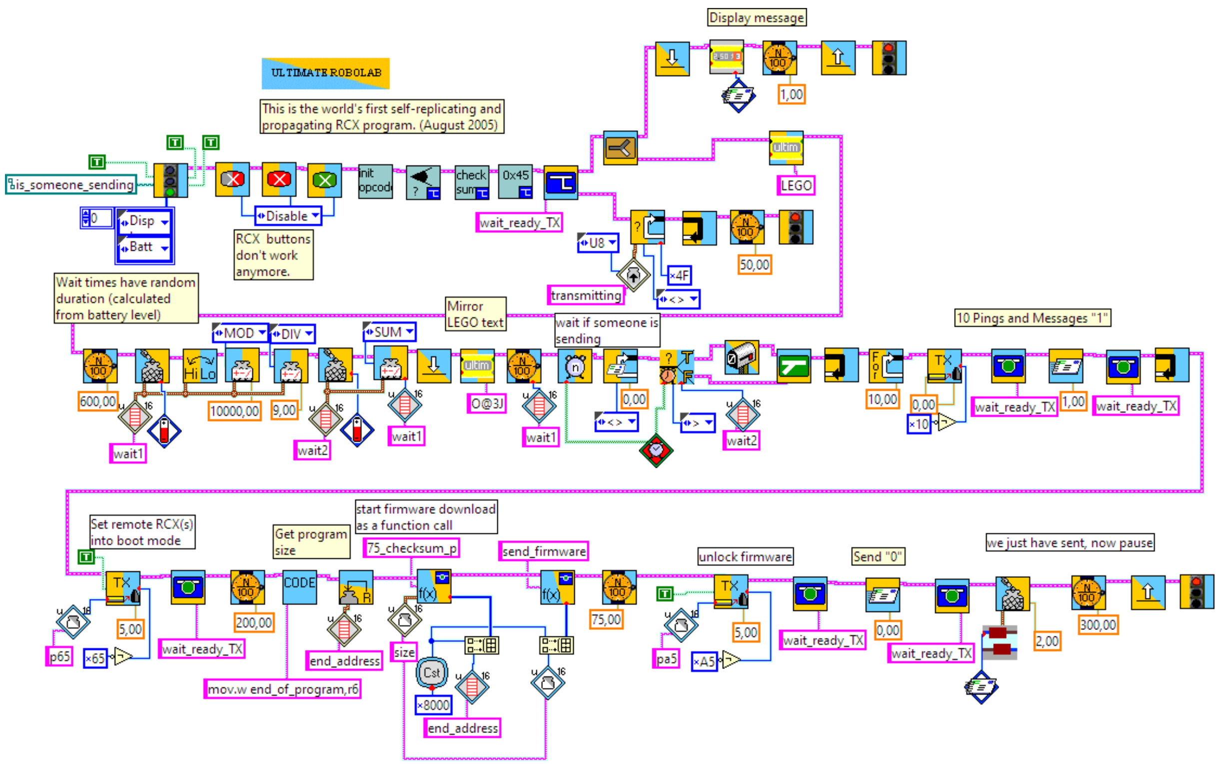


Figure 12: Main graphical program code of the virus firmware.

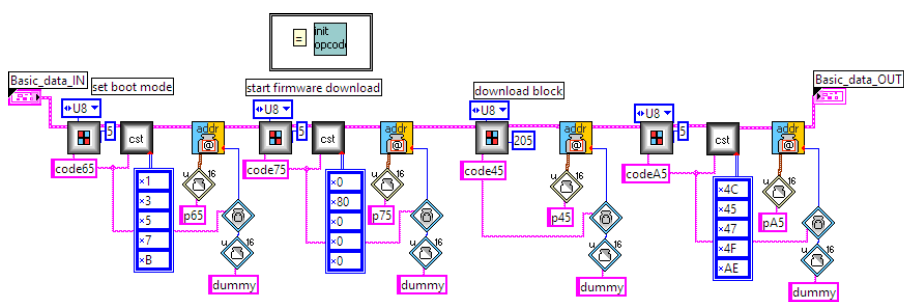


Figure 13: The opcodes to send are initially stored in arrays.

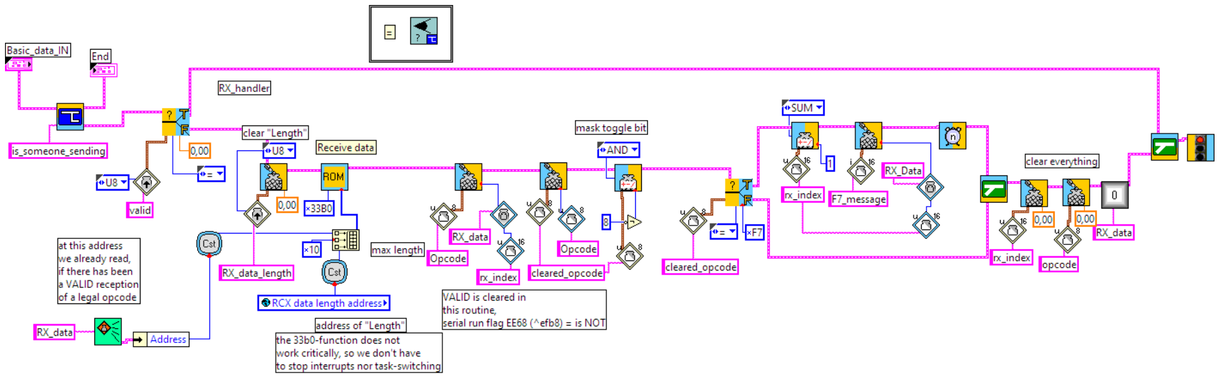


Figure 14: This subroutine is added to the background handler.

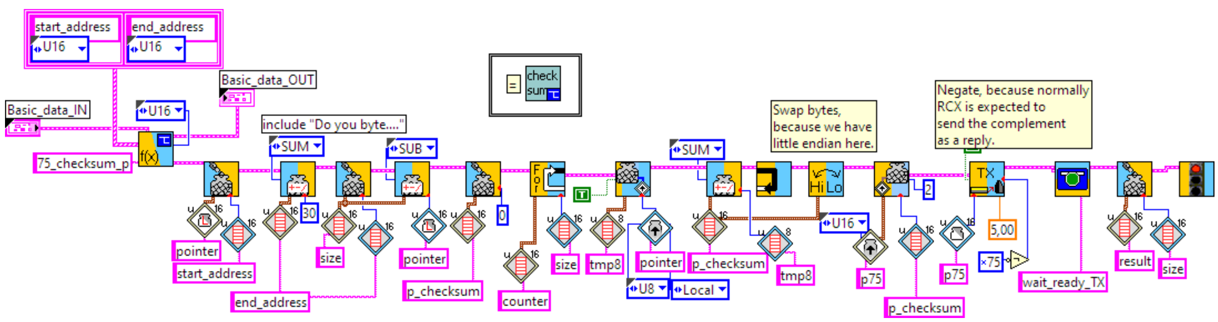


Figure 15: This function calculates the firmware checksum.

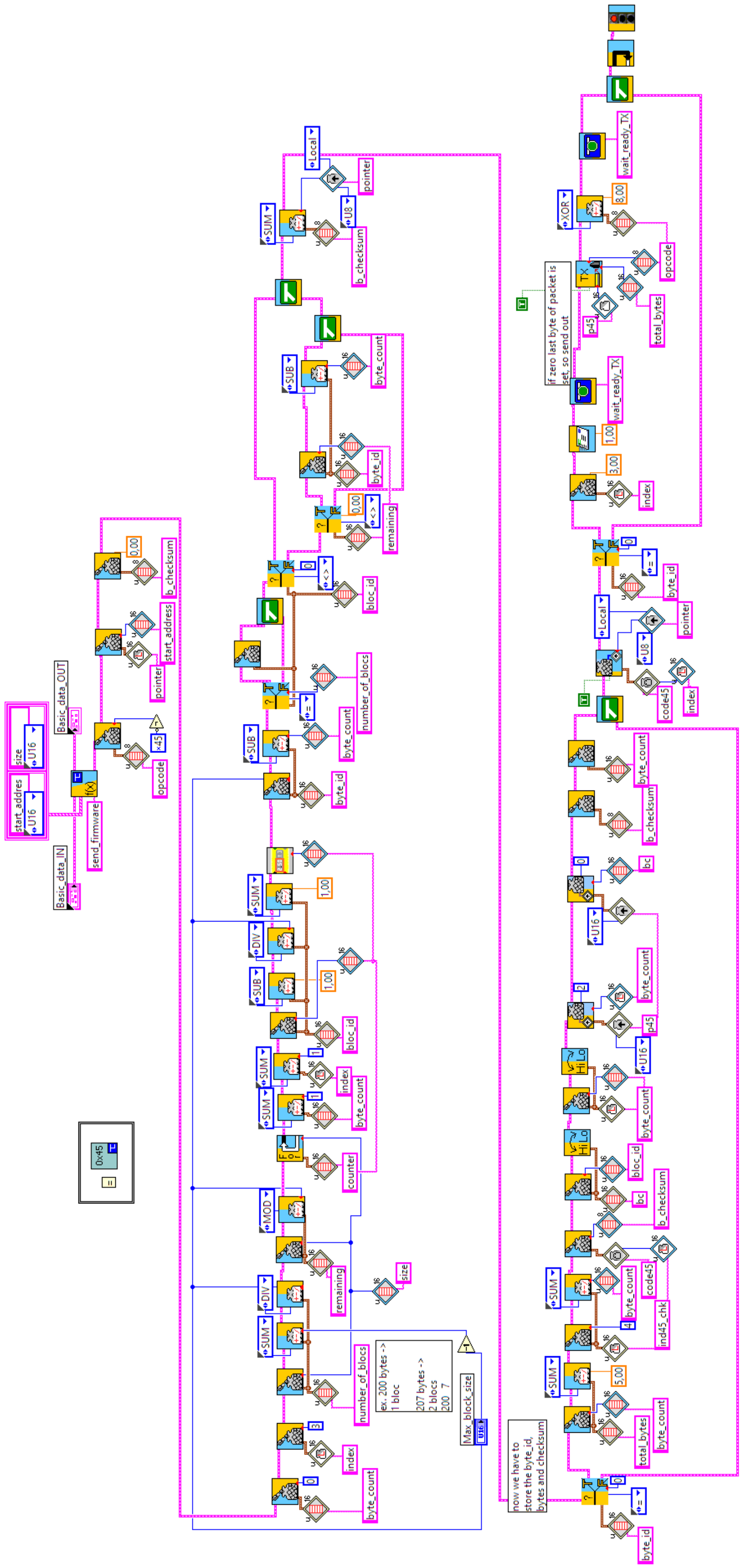


Figure 16: This function sends the firmware via successive opcodes 0x45.

15 Robot programming

The example of the RCX virus firmware demonstrates how efficiently the interplay of software polling, interrupt handling and multitasking management can be implemented into a tiny micro-controller like the H8/3292, proving the excellent choice of the micro-controller for becoming the heart of the RCX. The benefit-cost analysis, which its developers had made, undoubtedly dealt with an interestingly new type of problems, which consisted in discerning the fine differences between the general purpose computer and the new class of robotics controlling devices. From many points of view both systems are very similar, if not identical in terms of hardware and software architecture: I/O output management, program execution, interrupt handling, multitasking control, etc. For instance, if we consider the RCX from this perspective, we see a tiny computer with I/O features for human interaction: 4 buttons, mini-display and IR-communication with other devices.

However, even simple kid-designed robots require more than that! On the hardware level, there must be some kind of interface to the real (and real-time) world providing input/output connections for sensors and actuators of any kind... and –not to be forgotten– for physically mounting the RCX as a real object into a machine. In order to produce **autonomous robot behavior**, a multiple task system has to run well woven parallel state machines on the software level analyzing sensor input and controlling motors according to a higher level master plan. This adds new challenges to the embedded software design.

In this section we want to investigate, how a robust interrupt-based multitasking management can be set up within the guts, but also the limits of the RCX, and how this can be adapted for robot control.

15.1 Interrupts

All interrupts, except for the non-maskable interrupt ($\overline{\text{NMI}}$), are enabled/disabled together through clearing or setting the **I**-bit of the 8-bits **CCR**-register, which contains internal processor status information including the carry (**C**), zero (**Z**), negative (**N**) flags. Each single interrupt may be configured and enabled individually by configuring the corresponding device registers. (Refer to the documentation for more information.)

```
orc  0x80, CCR    ; disable all interrupts except NMI
..
andc 0x7F, CCR    ; enable all interrupts
..
```

(Note that pending interrupt conditions are not affected by changing the **I**-bit. Thus, if the interrupts are disabled, pending are not executed until the **I**-bit is cleared again. Sometimes it is therefore necessary to manipulate by software the relevant device registers, in order to clear pending interrupts.)

If an interrupt occurs –ass the word says– the H8/3292 interrupts the current process, stores the status register (**CCR**) and the program counter (**PC**) in the special memory space called the **stack** and fetches the address of the interrupt service routine (ISR) of the accepted interrupt with the highest priority (from $\overline{\text{NMI}}$ down to WDT) from the vector array array in ROM. In fact, in order to allow volatile firmware in RAM to use its own ISR, the RCX designers implemented a tricky method of interrupt handling. Instead of directly calling the ISR, the ROM vector points to an individual interrupt dispatcher code part:[14]

[14] Code snippet taken from a meanwhile disappeared web-page by Ole Caprani, University of Aarhus, Dk.

```
; H8/3292 interrupt vector address: address of interrupt dispatcher
interrupt dispatcher:
       push r6   ; save the contents of r6 to the stack
       mov @RCX interrupt vector address, r6
       jsr @r6   ; indirect jump to subroutine
       pop r6    ; restore r6 from the stack
       rte         ; return from exception
; RCX interrupt vector address: address of RCX interrupt handler
```

By this way, the user can define its own interrupt service routine (ISR) and store the address to the special RCX vector list (cf. section "RCX Hardware Portrait".) Because the dispatcher applies an indirect jump to the ISR subroutine (**jsr**) through hardware register **r6**, the contents of the **r6** must first be stored to the stack. At return, the initial value of **r6** must be restored from the stack (cf. Fig. 17).[15]

Because interrupts occur to any unpredictable instant, where the CPU is occupied with data manipulation, it is essential that before running the ISR subroutine, the rest of the sensible register data are saved to the stack, where they can be restored later, before the return from the subroutine. Note that the data from the stack must be restored in reversed order.

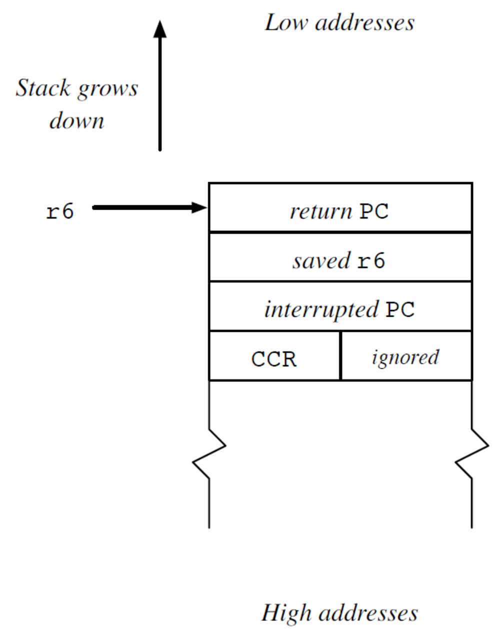


Figure 17: Interrupt action: the contents of the program counter **PC** and the status register **CCR**) are pushed to the stack. Note that we exchanged PC and CCR, because of a mistake in the original picture (Source: cf. footnote 15).

[15]https://www.classes.cs.uchicago.edu/archive/2006/fall/23000-1/docs/rcx.pdf, [retrieved November 2022], p. 4.

```
interrupt service routine:
       push r5   ; save the context
       push r4
       push r3
       push r2
       push r1
       push r0
       ..
       ; do something
       ..
       pop r0    ; restore the context
       pop r1
       pop r2
       pop r3
       pop r4
       pop r5
       rts        ; return from subroutine
```

It is essential to consider the interrupt latency, which is the duration between the moment, when the interrupt was triggered and the beginning of the ISR. Note that the instruction execution time of the H8/3292 varies from 2 (*add*), 10 (*rte* to 14 (*mulxu*) clock cycles. External memory *mov* instructions require about twice the time needed to access on-chip RAM, 2 .. 12 cycles. The average instruction duration can be estimated as ≈9 cycles, which is about 0.5μs at 16MHz. Latency time can therefore be estimated between 1 and 3.3μs. The interrupt response time, which adds the execution time of the ISR to the latency time, can be estimated between 20 and 120μs.

15.2 Multitasking

As we can see, timing is a critical part of RCX software design. The designers of the original RCX firmware made their choice for a 1ms interrupt handler (OCIA) that should act as the scheduler, besides processing some important control functions:

- Motor waveform update (pwm)
- Sensor power update (pwm)
- Enable A/D conversion
- 1/100s timer update
- Task switching

The RCX needs one particular process to run more frequently than normal user tasks. The purpose of this special task is to process some important updates that cannot be operated in the OCIA ISR, because of the risk of system instability. This background handler processes:

- Sensor update after A/D completion
- Refresh display (running man)
- Sound control
- Battery survey
- Button control
- IR received opcode handling

In order to keep the present description clear enough, we opted for explaining the multitasking system used in the ULTIMATE ROBOLAB software, instead of the original firmware. The main reason is that with the original firmware we have to consider serious version differences.

ULTIMATE ROBOLAB, the LABVIEW-based environment behind the virus firmware from the previous section, uses a preemptive multi-tasking system following the Round Robin method. Each task gets a 1ms time-slice, after which the next task is chosen. Additionally, the scheduler is not allowed to change the task, except for the background handler, in the case of short critical sections that should not be exited, because a certain robot behavior must be briefly maintained, or access to a commonly used resource cannot be currently shared without control issues. (Note that sometimes, task sections may even be hyper-critical, so that it is indicated to temporarily disable interrupts.) For some other reason, a specific task may have been inhibited, because a certain robot behavior is momentary undesired. The scheduler may therefore not switch over to such an inhibited task. Ultimate Robolab does not control waiting tasks. This must be seen as a flaw of this programming environment.

The major problem the task scheduler has to solve is to save and restore the task specific context at each task switching. As we have seen so far, interrupts –and this concerns the OCIA interrupt– save the hardware register context to the stack. The only thing that changes now is that the main return address has to be adapted, since the processor should jump to the next task code, instead of the one that has been shortly interrupted. If memory organization is such that no other variables are stored on the stack but rather in the global memory, no further variable data is part of the stack. However, it is more than likely that the task performed some function or subroutine call, so that other jump data has been placed on the stack. These program addresses are evidently part of the task context and must be saved in any case. ULTIMATE ROBOLAB solves this issue by storing the task specific **stack pointer (r7)** to a data array, from where it can be fetched, if required. Additionally the stack is divided into sectors that are reserved for each task. The user has to take care that no stack overflow happens, which would produce fatally uncontrolled program behavior.

The only thing that ULTIMATE ROBOLAB surveys beyond this description is a security measure, in the case that all the tasks have been inhibited, which represents a deadlock situation. If this happens, the system would hang and freeze. The user has to make sure that no task stays in a critical section for a very long time, because this would cause task starvation.

15.3 Critical section protection

Starting from this low kernel level, abstraction can be pushed much further. That's exactly, what ULTIMATE ROBOLAB was designed for, making possible the implementation of valuable protections of commonly used resources or critical program sections. In fact, ULTIMATE ROBOLAB allowed the use of the higher end semaphore method as defined by computer pioneer Edsger W. Dijkstra.[16]

Why is the fact that more tasks use resources in common an issue? The easiest way to explain this, is to have a practical demonstration. If we have a look at the **Ultimate Robolab** program of Fig. 18, we see that both parallel tasks have access to the sound channel at the same time. Running this firmware results in a badly mixed music tune, a real cacophony, because one task overwrites the control data of the other.

[16] cf. for instance https://en.wikipedia.org/wiki/Semaphore_(programming), [retrieved November 2022].

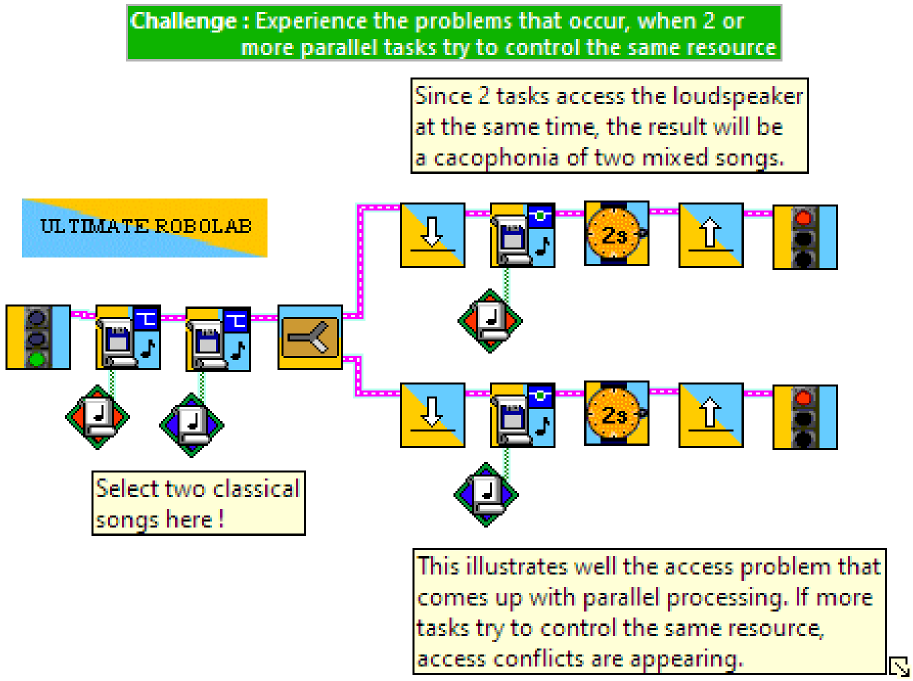


Figure 18: This program produces cacophony, because two parallel tasks control the sound channel at the same time.

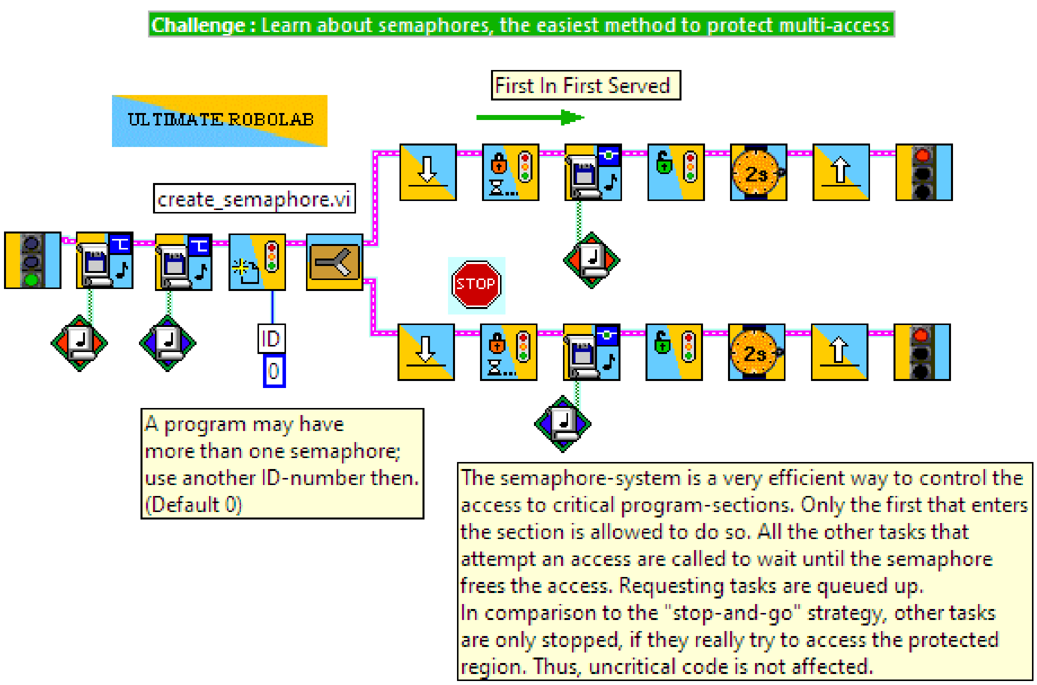


Figure 19: This program solves the resource collision issue. Semaphores guarantee mutual exclusion. The second task waits until the song has terminated in the first task and vice-versa.

There are many solutions to this kind of issue: task prioritization, task blocking, etc. However, probably the most efficient method of mutual exclusion is the implementation of a semaphore system. Note that the code for such a system can only be designed in Assembly language with great difficulties. At this point, higher level languages must do their job. Fig. 19 shows the programming ease that can be obtained by higher level abstraction for such a system. The first task only frees the resource after the song completion by opening the semaphore locker. because the second task is in the waiting queue, it accesses the sound channel after it has been freed by task 1, and blocks it for the concurrent task, which now has to wait for song completion. Visibly the semaphore method follows the First-Come-First-Serve (FCFS) mechanism. The first task that tries to enter its critical section has the priority of action. The requests of all other tasks trying to get access are added to a queue, which is rewound in the sequel.

15.4 Subsumption architecture

Although the firmware features that have been presented so far are already very performing, it must be said that they still do not fulfill all the requirements for efficient robot programming. Even if we add event-handling and finite state-machines, priority control to the features, the implementation of layered robot behavior into a limited embedded system remains flawed, because program design may become excessively complex. It certainly goes beyond the scope of this paper to fully develop the basis of a conflict-free architecture needed for robot programming. However, we want once more underline the remarkable design of the RCX, because it allows the implementation of the **subsumption architecture**, which at the moment of this writing still is considered the most valuable program structure for the large class reactive robots. The method has been invented in the 80s by Rodney Brooks, former MIT professor and founder of the IROBOT CORPORATION.[17]

Robot architectures have to solve very specific problems. Because robots operate in the real world, they have to:

- control actuators to respond to real-time sensor input
- react on sensor stimuli with determined higher-order behaviors
- find valuable reactions and solutions in unexpected situations
- follow some specific goals
- continuously survey that the system runs as expected
- solve the conflicts that may result from concurrently managing all these functions

As said, we only can give a glance to the complex field of robot architecture here. Fig.20 shows the use of a subsumption architecture kernel within the limited RCX. The program starts with the definition of all the required functions including the behavior arbitrate. Follow behavior definitions: move forward, turn right, turn left. Then, in the second row, the program starts the cruise function, which is the default behavior here at lowest priority. After this icon, two buttons are configured as touch sensors. Now there are two concurrent tasks reacting on sensor input. The lozenges indicate that the task requests a defined behavior with different priority. The arbitrate then chooses the highest priority behavior to be executed and returns to lower priority behaviors after completion. This is marked by the request **NONE** to the arbitrate. The control architecture doesn't stop or block any task, but only inhibits the access to the resources, actuators in this case. This allows the robot to continue seeking for new stimuli. Note that the resulting program obviously does not represent the most efficient control for a two-sensor wall-avoider, because the robot could get stuck in a corner. This is just an example of how a conflict-free program can be implemented using higher level programming abstraction using the subsumption architecture... and all this within the insignificant but gigantic LEGO RCX!

[17] R. Brooks, *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, Vol. 2, No. 1, (1986), pp.14-23, cf. https://people.csail.mit.edu/brooks/papers/AIM-864.pdf, [retrieved November 2022].

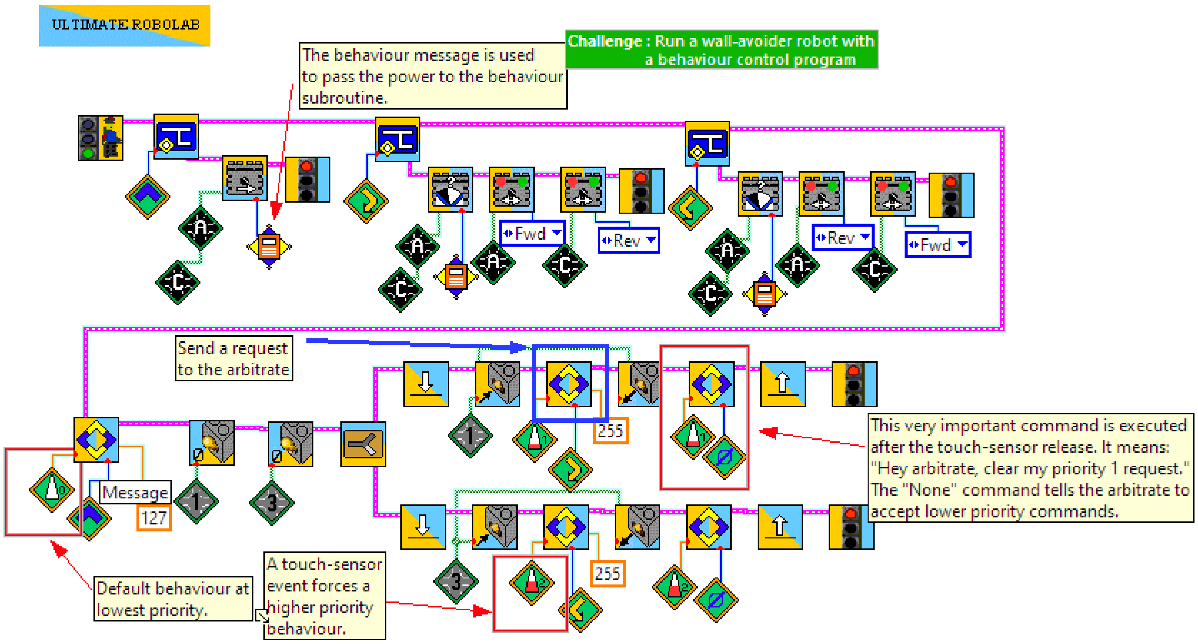


Figure 20: With this firmware, the RCX controls a wall-avoider robot with an implemented SUBSUMPTION ARCHITECTURE.

16 Conclusion

Of course this document only gives a microscopic view on the LEGO RCX, given the huge number of Internet site, books and articles that have been composed on the subject. However, the purpose of this paper has been clearly dual: describing how to reactivate the device with modern computers and gathering some fundamental information absolutely required for the vintage computer preserver. Perhaps, some museum conservator might eventually develop an interactive show with good old RCX , instead of just exhibiting the precious device in a display case. What a joy it would be to see a revived robot activity like the unsurpassed Synthetic Jungle Cube Project by Prof. Ole Caprani, University of Aarhus, Dk, (2000).[18].

---

[18]https://cs.au.dk/ ocaprani/legolab/Projects/JungleCube.dir/