Document release 06/03/22

The ELLO Computer Project

1 Introduction

What is ELLO about?

In one sentence – to have fun building and learn in the process.

The focus is away from competing with the big guns, such as Raspberry and the likes, but rather an alternative option to enjoy building and programming a small and very unpretentious computing system, entirely from scratch.

So, what are the goals?

When started with the project, I set several important goals ahead:

1. **To let the user be able to build the entire computer themselves.**
   This means to be as simple as possible hardware, built only with atomic components (no pre-assembled modules), and without significant compromises on its basic functionality.
   As "basic functionality" I defined the following set of parameters:
   a) To be able to use standard input and output devices – keyboard and monitor.
   b) To have some available non-volatile storage.
   c) To have some reasonable minimum amount of RAM.
   d) To have some means to produce sound.
   e) To have some expansion capabilities.
   f) To be built only with parts which are large enough for handling even by the most inexperienced user. This significantly complicates the choice of components as it excludes almost every available modern microprocessor or microcontroller as they are all manufactured only as small surface-mount packages.

2. **Not to use a heavy and resource consuming operating system.**
   Trumpeting about great hardware and then leave the user banging his or her head in the wall, and having to search the Internet every time they need to write something, gives no pleasure to the user nor teaches them anything other than developing proficiency in using search engines. Hence, anything including the word "Linux" falls automatically into the "no-no" category of choices.

I decided to make the ELLO in the same proven way as the early personal computers – a built-in interpreter to allow immediate integration with the system.

3. **To offer a "meaningful" programming language.**
   The opinions what programming language should such small computer include, vary a lot indeed. About 99% of all similar systems today run some dialect of BASIC, which by itself is an excellent and powerful programming language.

   In contrast to all, ELLO includes a tiny C interpreter!

   In my view and with the full respect to BASIC, which was my first, just like to many others, C is the one truly "universal" programming language, and worthy to be learnt by anyone who wants to have a closer contact with programming. It is the de-facto "The Language" in the modern computing world, especially in the modern embedded computing world. C is also much more "unsafe" language with significantly lower tolerance for errors than BASIC or other alternatives, especially when working with direct access to the memory and the other hardware resources. While that might sound scary, it is also beneficial for the user to learn and work without having an error net which saves them from mistakes otherwise. Teaching careful programming and awareness of the system resources can do only good to any programmer.

   Due to its powerful features and loose syntax, C is extremely difficult for implementation as a runtime interpreter – only a handful ever written, none of which able to work well in a small bare-metal OS-less system like the ELLO. I jumped on this additional challenge with great enthusiasm, but it still took many months of hard work and debugging to get it to the current level.

   Of course, the C dialect running on the ELLO is much simpler than its typical desktop equivalents, but still includes all the original language elements and even adding some limited compliance with the C99 standard. Its execution speed is of course far from what a compiled program would be like, but it still serves quite well its purpose in this case.

   Finally, I owe big thanks to Geoff Graham for the source of his "ASCII Video Terminal" which was a valuable reference for me for the video generation and support of a PS/2 keyboard, and saved me many hours of development.

Also, extending the thanks to Alan Ott for his excellent and simple to use open-source “M-Stack” which ELLO uses for the handling of USB console.

2 Hardware

2.1 Specifications

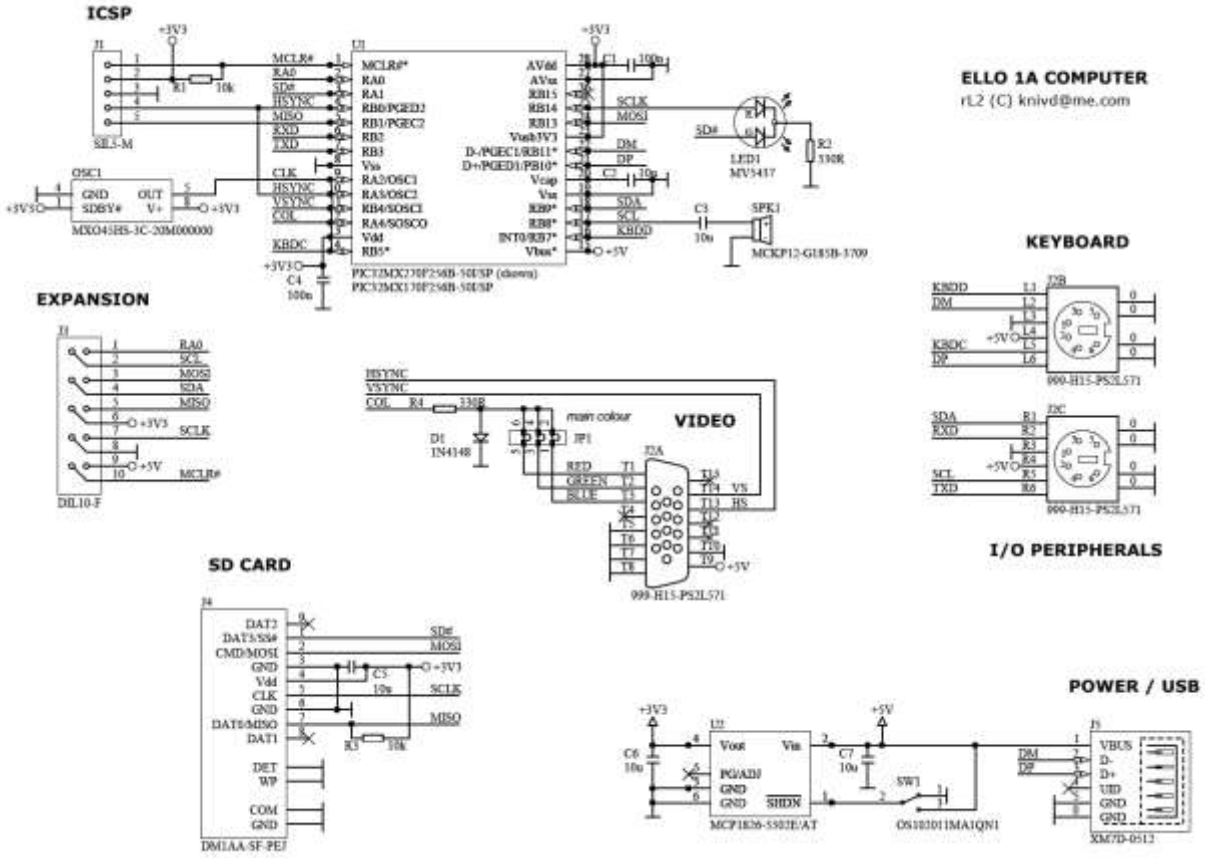
Model:	ELLO 1A PCB revision L2
Chip:	PIC32MX170 or PIC32MX270
Frequency:	50 MHz
Total RAM:	64 Kbytes 12 Kbytes reserved for system needs and stack 15 Kbytes reserved for video page 37 Kbytes memory available for user's programs
Total ROM:	256 Kbytes 194 Kbytes system software 62 Kbytes available as non-volatile storage drive
Video:	VGA, 7 selectable main colours 480 x 250 graphic pixels *(480x320 in earlier versions)* 80 x 25 text characters *(80x29 in earlier versions)* ***NOTE:** 480x250 is not a standard resolution, so you may need to play a bit with the controls of your monitor*
Sound:	On-board speaker
Input:	PS/2 keyboard connector
Storage:	Small internal flash storage space - drive **IFS:** Full size SD card holder - drive **SD1:**
Expansion:	Mini-DIN connector with A/I/O, UART and I^2C interfaces Standard 0.1" header with A/I/O ports, SPI and I^2C Optional USB interface (with PIC32MX270 only)
Power:	5V DC through standard mini-USB connector Average current consumption 44 mA

2.2 Schematic

ELLO 1A is among the simplest computing systems, built from only 25 components, some of which optional. It is in fact so simple that a prototype can be easily built on a breadboard. The PIC32 microcontroller is a single chip in the heart of the computer. It handles everything in software – the video, the interfaces, sound, and executing the C interpreter. The reason I chose this particular model is, because at the time of the design it was the most powerful microcontroller on the market, coming in a through-hole package. It is still quite limited in comparison with other modern chips, but at the same time is a completely reasonable option for the main goals of the project.

I have intentionally designed the schematic in such absolutely minimalistic way so it can be easily understood by anyone.

As an additional note, the diode D1 is to provide voltage level protection on the monitor's input. All monitors though, include their own protection on the input, so installing D1 on the board is optional.



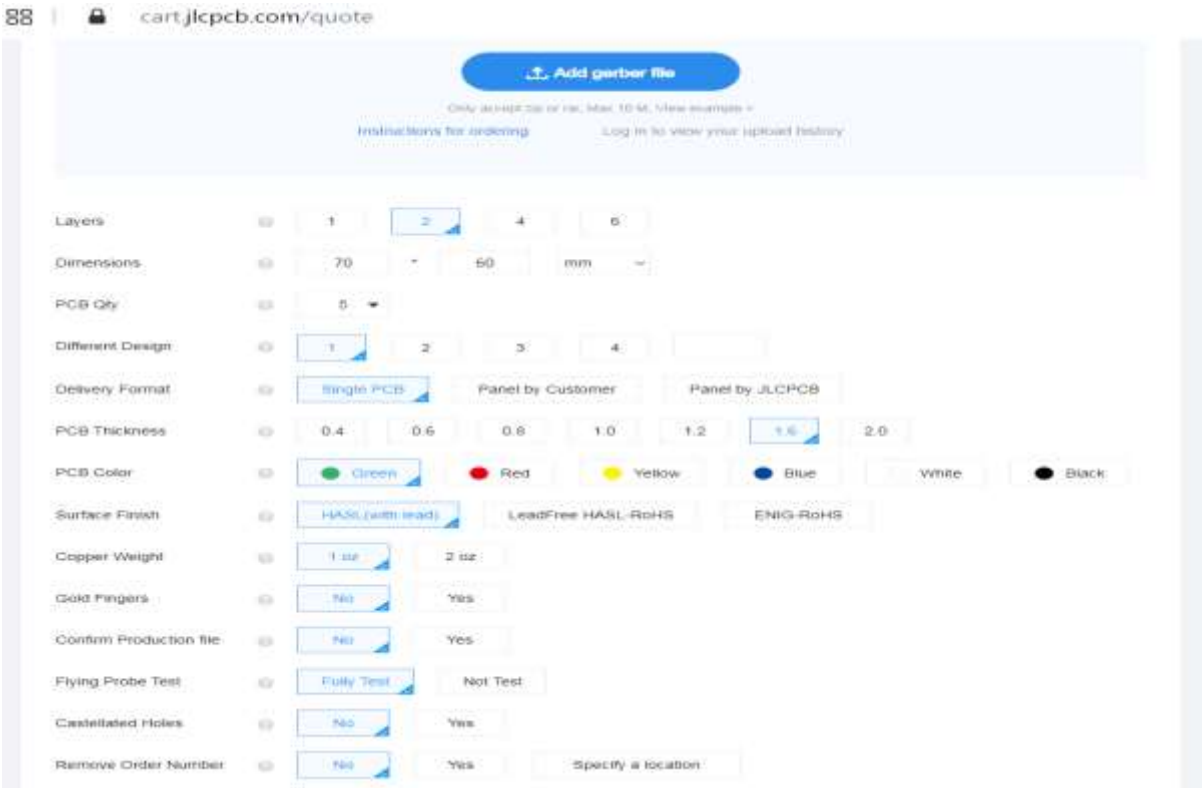
Pursuing maximum simplicity and the limited number of input/output pins on the microcontroller come at the expense of solutions which are not exactly “by the book”, and may make some hardcore engineers feel uneasy. One of the great advantages of the PIC32 is its sturdiness and tolerance toward imperfect environments, which works great for the purpose of this project.

2.3 PCB

ELLO 1A is built on a small 2-layer PCB with size 70 x 60mm (2.76” x 2.36”).

I order the prototype boards from JLC: https://jlcpcb.com

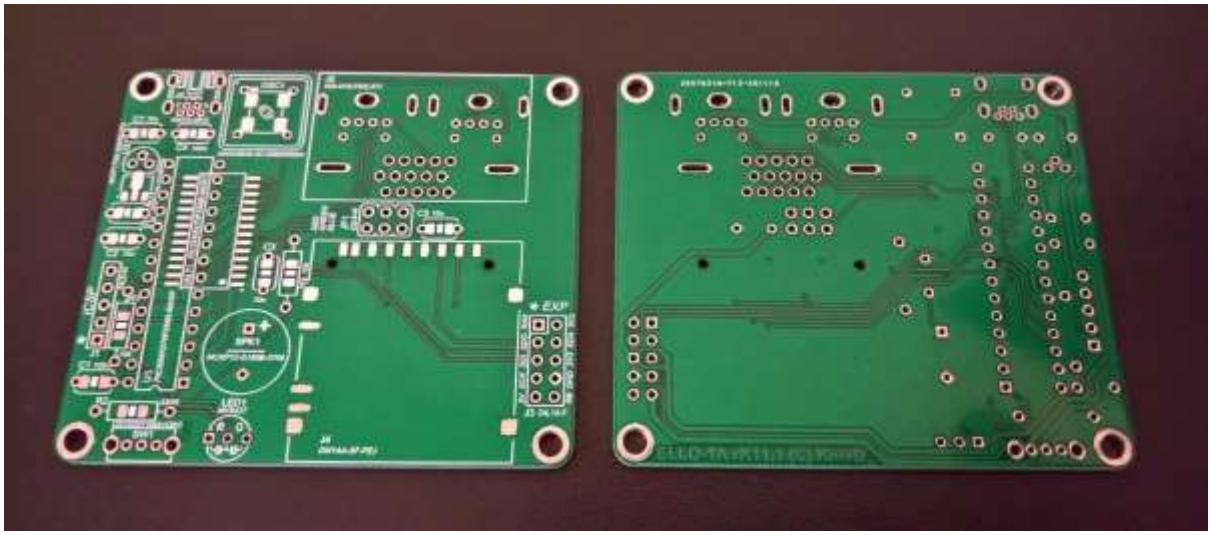
Their quality has been great in all boards I have ordered so far, and the prices are unbeatable. When ordering, the default parameters suit just fine for this board.



The PCB is designed in such way, so the component footprints take either the through-hole or the surface-mount (or a mixture of the two) BOMs, according to what parts are available or preferred by the user. There are a few exceptions, though, such as the VGA/PS2 combo.

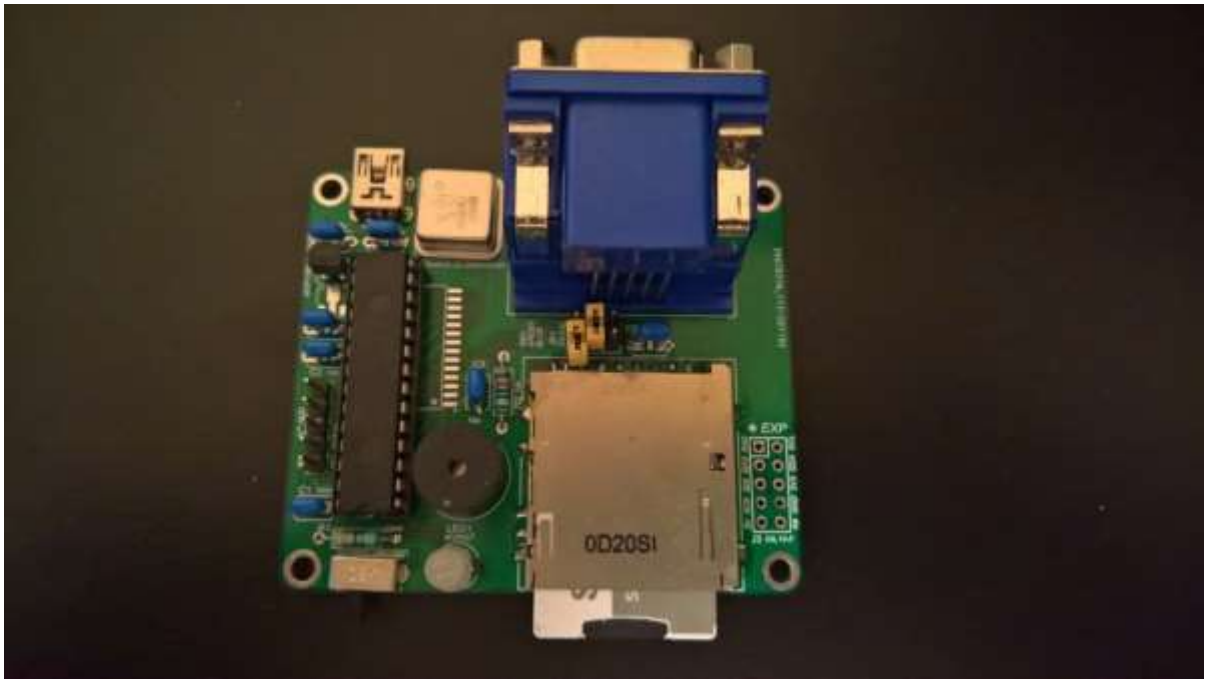
For the assembly, I only used my small soldering iron. It takes about an hour to solder all the components without rush.

My recommendation is always to start with the SD card holder in a through-hole assembly, and with the PIC32, followed by the power supply IC and the SD card holder, in a surface-mount assembly.



Surface-mount components are best soldered when the board is perfectly flat, so ideally they should be preceding any through-hole components.

Whenever possible, I recommend installing a socket for the PIC32 chip in the through-hole variant. This recommendation has been reflected in the TH BOM as well. The large VGA/PS2 combo connector should be installed last, and the PIC32 chip plugged into the socket only after the entire board has been assembled.



3 Software

The built-in software in ELLO 1A can be roughly divided into three main parts: system functions, **RIDE shell**, and **C.impl interpreter**.

The system functions are a set of various hardware-dependent functions, completely transparent to the user, and dealing with the overall operation of the hardware, such as producing the video signal, various timing routines, handling the file allocation system on the storage drives, etc. These are the only parts in the entire firmware, which are hardware-dependent.

3.1 The “RIDE” Operating Environment

RIDE (“*Rationally Integrated Development Editor*”) is where the user interacts with the system. In essence, it is a line text editor with built-in extra commands for file operations and system control.

When started, an access password is required in order to let the user to use the system. By default, there is no password, so a simple **<*Enter*>** key is all that is

needed. The user is able to set their own password at a later stage later from within RIDE, if one is needed.

An important detail to start with - RIDE is not a typical editor but a line-based one. This means that what is shown on the screen is not necessarily how the text actually looks in the memory. Individual lines can be displayed, edited, and then other lines displayed below them in a non-sequential order. Using a line editor may seem confusing at first, especially to those who have never used one before, but also brings several benefits. First, it does not depend on the hardware in any way. The line editor will look the same way on a small 2-line LCD as it will look on a large terminal screen. Display height and width don't matter. Another benefit is, once mastered, using a line editor actually could help make writing code easier.

For example, in a source of several hundred or several thousand lines of code, a developer might find themselves frequently scrolling up and down over large chunks in order to check and refer to different parts of code. In RIDE this is achieved by typing short one-character commands or sequences. In a short 'dot-command' line the developer could for example jump to a line and change something, then jump elsewhere change something else, and then jump back to the initial location. RIDE also supports repeating the same command defined number of times. This is useful when performing search and replace operations in the text.

Commands to the editor are entered starting with a '**.**' character from the first position in a line. A single dot is sufficient for an entire command string. Spaces are in the command string are ignored.

Editing could be made in a script-like fashion by giving several commands at once to RIDE. For example, typing the command string "**.S20 J55 C5**" will tell RIDE to set line 20 as source from which lines can be copied or moved, then jump to line 55, and copy 5 lines there. In result lines 20, 21, 22, 23, and 24, will be copied starting from line 55 on, and lines which were originally following line 55, will be pushed further down.

There are several commands, such as those expecting a file name, which don't allow more RIDE commands to follow on the same line, because the entire rest of the line is assumed as parameter.

A full help of all commands can be obtained by entering the **.H** command. Information about the current status of the editor can be seen by entering the '**?**' command.

.h

>>> <Ctrl-Z> History <Ctrl-N> Next line or add new
>>> <Ctrl-L> Clear full line <Ctrl-Y> Clear to end of line
>>> <Ctrl-V> Clear screen <Ctrl-U> Print line ruler
>>> '.' at start of a line marks a command line
>>> '@' represents the current line N '#' source line N
>>> '!' is the number of lines incl current and to the end
.^ code Set keyboard break code (default 3)
.N pw, ph Page width and height (12-999 chars, 2+ lines)
.T width TAB width (1-80)
.~ Equivalent to <Del> .\ New line <Enter>
.* count Repeat the following cmd for spec'd number times
.`text` Insert txt at the cursor pos in the current line
.X [hpos] Place cursor at spec pos within the current line
.[J] [< or >] [number] Jump to line/prev/next; '.J' last N
.L [[P]count] List next N lines or until EOF [option Pause]
.P [[P]count] List previous N lines or from start [Pause]
.I [count] Insert blank lines. One line if [cnt] is missing
.D [count] Delete lines. Only the current with no parameter
.S [line] Set source line for copy and move ops .H .?
.C [count] Copy lines from source to current. Default num=1
.M [count] Move lines from source to current. Default num=1
.F [`text`] Set text for find; w/o param, do find <Ctrl-F>
.R [`text`] Set text for replace; w/o, find&repl <Ctrl-R>
.E file Execute command script from file
.= [file] Run source in memory or from file
.O file Open text file .W file Write .A file Append
./init drv: Initialise drive ./lock pwd Lock access
./dir [path][mask] List files. Accepted wildcards '?', '*'
./chdir [drv:][path] Change drive and/or dir; Show current
./mkdir path Make new dir ./rmdir path Remove empty
./del mask Delete files ./ren mask_old , mask_new
./copy mask, mask Copy files. Accepted wildcards '?', '*'
./list file List text file ./blist file List bin file
./date YYMMDD Set date ./time HHMMSS Set time :24h
./reset System reset

In the text above, the parts enclosed in [] brackets are optional. Hence, a the most often used command ‘jump’ can be in the form of .J followed by an optional line number, or just a dot followed by the line number. A single .J command with no line number to follow will jump to the last line in the source.

As an example, the command ‘**.JP**’ will list the entire text from the start. The same effect could be achieved by executing ‘**.1L**’ too, with the difference being that in the first case the cursor will remain at last line in the source, whereas in the second, it will remain at the first. Adding one more ‘P’ to the form ‘**.JPP**’ will ensure the listing pauses properly at every page.

Commands for copy and replace require a ‘source’ line which marks the start of a block of one or more lines. The operation then takes from the source and copies or moves to the destination at the current line of the cursor.

Let’s write a very simple test program in C:

```
1: #include <stdio.h>
2: for (int i=0; i<3; i++) printf ("Hello, World!\r\n");
3: |
```

Let’s now execute it by typing the command ‘**.=**’, followed by <*Enter*>, in the current empty line:

```
  1: #include <stdio.h>
  2: for (int i=0; i<3; i++) printf ("Hello, World!\r\n");
.=

Hello, World!
Hello, World!
Hello, World!

  3: |
```

A program in the memory can be cleared with the command **.1d!**

Dissecting what the command actually does – it jumps to the first line, and then deletes the number of lines from there to the last one.

Other fundamental commands in RIDE are **.o** to open and load a file, and **.w** to write a file.

A file can be directly executed without loading into the editor with command **.=**
For example, **.ohello.c** will load the file "hello.c" in the editor, while typing instead **.=hello.c** will directly run the program.

To demonstrate some of the RIDE's capabilities, here is another example.
Let's consider writing this generic text:

1: This is line 1
2: This is line 2
3: This is line 3
4: This is line 4
5: This is line 5
6: |

Let's to add '00' in front of all numbers. This command will do the job: **.1*!x6`00`>**
After execution, the result will be
6: 00|

This is a natural result of the execution. Just delete the excess characters by pressing Ctrl-L, and list the text: **.p**

1: This is line 001
2: This is line 002
3: This is line 003
4: This is line 004
5: This is line 005
6: |

If we now want to restore how it was before: **.1*!x6~~>**

RIDE is capable of executing complex command sequences and even have them executed as script from a file.

3.1.1 Storage Devices

There are two storage devices included in ELLO 1A. The first one is **IFS:** and is built within an area of the internal flash in the PIC32. The second is **SD1:** and refers to the SD card slot in the system.

IFS: is a very small (around 70 Kbytes) drive, whose main purpose is to store one or two executable files and some configuration data in an embedded manner. It is also prone to flash wearing off as result of continuous writes into it, so using the IFS: drive should be limited to only occasional writing.

On power-up, RIDE looks for a file called '**IFS:/AUTORUN**' and if such exists, executes it. If an auto run file is not found in IFS:, the SD1: drive is searched for one, too.

Thus, the user has the option to enable automatic execution of a program every time the system starts.

Upon entering the editor, if card is found in the **SD1:** slot, it is made current drive, otherwise **IFS:** remains as current drive.

3.1.2 File Operations

RIDE supports basic file operations such as listing a directory of files, making and removing directories, copying, deleting, and renaming files, as well as formatting a new drive (the command '**./init**'). The file system used is FAT16 or FAT32, depending on the size of the drive, and file names are in 8.3 character format.

File commands support wildcards '**?**' and '*****'. For grouping more than one file in an operation. For example:

A command '**./ren C*.C, D*.C**' will rename all files whose name starts with 'C' and have file extension .C, to the same file name but starting with 'D', instead.

A command '**./dir TEST??3.TXT**' will list only the files with extension .TXT whose name starts with 'TEST', have two other letters, and then finishes with '3'.

Additionally, RIDE includes commands to list the content of files in text or hexadecimal form.

3.1.3 Console mode

In some cases a user may have the need to communicate with the ELLO computer via a serial console. Such functionality is supported by the system software, and provided on the same port pins as the PS/2 keyboard, so both are mutually exclusive. On power up the system checks whether there is a PS/2 keyboard connected, and if there is no keyboard detected, switches automatically in console mode.

The console mode uses a fixed serial protocol (TTL signal levels) on the purple mini-DIN connector, normally allocated for a keyboard.

Just like the keyboard, a serial console uses the same pins 1, 3, and 5 on the connector for communication.

Pin 1: PS/2 keyboard data, or TxD channel from ELLO to the console
Pin 3: Common ground 0V
Pin 5: PS/2 keyboard clock, or RxD channel from the console to ELLO

If the system is built around a PIC32MX270 chip, serial console is also available on the USB channel.

The serial console works with a fixed protocol 38400 bauds, 8 bits data length, no parity, 1 stop bit (otherwise said “8N1”). The speed is chosen as a reasonable compromise to accommodate the vast majority of possible terminal devices connected to the ELLO.

The serial console works only with text output, and has no defined size for the screen. The video output from the system is still available, provided there is a monitor connected on the video port.

A system with no display gains about 15% in increased computational performance due to the fact the processor is freed from the tasks associated with generation of a video signal. It also makes use of the reserved video RAM, so there is more memory available for user programs.

3.2 The “C.impl” Interpreter

C.impl means “*Simple*”. The aim is to achieve a close C90 specification functional implementation.

There have been countless number of books and other online and offline introductory and informational material already written on the topic of the C language. Repeating all that material here would be pointless, so I will present the C.impl capabilities only in a brief form.

First, C.impl is written in the way so it is a pure “execution-in-place” interpreter – there is no conversion from source into a tokenised stream, and the entire process runs in a single pass. While there is an execution speed penalty to this, a benefit is the possibility to run sources directly from a read-only memory.

An additional important detail to the fact C.impl is an interpreter – since there is no compilation and linking process, there is no pre-processor. The support for pre-processor commands in the source is reduced to only ***#include*** for inclusion of libraries.

//-style source commentaries are supported in the source, along with the standard **/*** … ***/** model.

C.impl distinguishes and supports both pointer to constant (eg. const char *) and constant pointer (eg. char const *).

One other important difference between C.impl and the C compilers, is that while main() function is supported and acted on properly, in C.impl it is not required, and if missing, program execution starts from the first source line. Hence, C.impl code may be executed outside of any function body.

3.2.1 Data Types

bool	1-bit integer
char	8-bit signed or unsigned integer
short [int]	16-bit signed or unsigned integer
int	System-specific width integer (32-bit in ELLO 1A)
long [int]	32-bit signed or unsigned integer
long long [int]	64-bit signed or unsigned integer
float	32-bit floating point

double	64-bit floating point
long double	80-bit or wider floating point *(supported in C.impl but not supported in ELLO 1A)*

In addition to the basic types above, several other data types are also supported

size_t	Natively represented as ***unsigned int***
enum	Standard enumeration list, evaluated as ***int*** values
struct	Standard C-type data structure
union	Standard C-type data union
va_list	In conjunction with **<*stdarg.h*>** for argument lists
FILE	File handler for standard **<*stdio.h*>** library functions

All integer types are signed by default (as if preceded by a ***signed*** declaration).

3.2.2 Data Constants

'character'	A single 8-bit ASCII character

Some non-printable characters (those with ASCII code smaller than 32) have predefined constants:

'***0**' - ASCII code 0 (character NUL)
'***a**' - ASCII code 7 (alarm)
'***b**' - ASCII code 8 (backspace)
'***e**' - ASCII code 27 (escape)
'***f**' - ASCII code 12 (form feed)
'***n**' - ASCII code 10 (new line)
'***r**' - ASCII code 13 (carriage return)
'***t**' - ASCII code 9 (horizontal tabulation)
'\\' - the character 'backslash'
'\'' - the character 'single quote'
'\"' - the character 'double quote'
'***?**' - the character 'question mark'

In addition to the predefined constants, any ASCII code could be also entered in its digital form:
‘***x***nn*’ with ‘*nn*’ as a two-digit hexadecimal code in the range 00…FF,
or as
‘*nnn*’ with ‘*nnn*’ as a three-digit decimal code in the range 0...255.

"string"	A zero-terminated string of 8-bit characters String constants can be split on to multiple source lines as long as there is nothing else other than whitespace characters between them

up to 64-bit decimal integer numbers; *optionally preceded by ‘**0d**’ or ‘**0D**’*
up to 64-bit hexadecimal unsigned numbers; preceded by ‘***0x***’ or ‘***0X***’
up to 64-bit binary unsigned numbers; preceded by ‘***0b***’ or ‘***0B***’
up to 64-bit octal unsigned numbers; preceded by ‘***0***’

[***sign***] [***iii*** [***.fff*** [***E*** or ***e*** [***sign***] [***eee*** [***.xxx***]]]]]

A floating point number

Where:
iii.fff are the integer part and the fraction of the number, accordingly.

The ‘***E***’ or ‘***e***’ symbol indicates that an exponent is being supplied to the number.

eee.xxx are the integer part and the fraction of the exponent, accordingly.

Numeric constants also support suffix specifiers for sign and/or size

Unsigned integer	Trailing ‘***U***’ or ‘***u***’
Long integer or long double	Trailing ‘***L***’ or ‘***l***’
Long long integer	Trailing ‘***LL***’ or ‘***ll***’

Single precision FP number	Trailing '***F***' or '***f***'
Double precision FP number	Trailing '***D***' or '***d***'

The default type of all integer constants, unless explicitly specified, is ***signed int***.

The default type of floating point number constants, unless explicitly specified, is ***float***.

3.2.3 Built-in Libraries

C.impl includes built-in several most used standard C libraries. A good reference to the standard C libraries can be found at https://www.cplusplus.com/reference/clibrary/

The useful command './lsl' in RIDE allows listing the pre-installed system libraries or the content of any one of them. When listing a system library, it is important to supply the parameter together with the enclosing <> characters. Thus as an example, './lsl <stdio.h>' should be executed in order to list the content of the stdio.h library.

<stdlib.h>	Library with standard general utilities. C.impl also adds: ***unsigned long BIT(unsigned char n)*** Return an unsigned integer with the specified bit number raised. Bit numbers start from 0, which is the lowest meaningful bit in the number.
<stdio.h>	Standard input/output functions. C.impl also adds: ***void run(const char *filename)*** Load and run a specified file. The new file is loaded in memory over the currently executed program.
<stdint.h>	Standard integer definitions.
<stdbool.h>	Boolean definitions.

<stdarg.h> Support for functions with variable number of parameters, provided by the ellipsis (…) operator.

<string.h> Functions for manipulation of zero-terminated strings and memory transfers.

<math.h> Mathematical functions and constants.

<limits.h> Definitions of various numerical limits.

<ctype.h> Functions for check and conversion of text characters.

<time.h> Library with data types and functions for timing.

<assert.h> Runtime assertion of an expression. C.impl also adds the possibility assert the existence of installed library functions, eg. *assert(printf());*

The library is also expanded with the following:

void error(int code, const char *message)

Trigger an execution error with supplied exit code and optionally display a message. If a message is not required, then the parameter is allowed to be NULL.

In case the error code is 0, the program will terminate as if it has correctly finished execution.

In order to keep backward compatibility with future versions, the error codes should be always negative numbers.

Error codes between -1 and -99 are reserved for C.impl and should not be used.

In addition to the standard libraries above, these non-standard ones are also included:

<conio.h> Functions for work with a generic I/O console.
The <conio.h> library is to a large extent a subset of the <stdio.h> library with a notable exception being the kbhit() function, not present in <stdio.h>.

Another important difference is the behaviour of the putchar() function, which in <conio.h> works exactly as printf() with consideration of the control codes, while the same in <stdio.h> always outputs the visual character of a given code.

<fatfs.h> C.impl wrapper functions for most of the FatFs library.

	FatFs is a popular open-source library for work with files and drives in FAT16 and FAT32 disk volumes. Full information about FatFs is available at the developer's website: http://elm-chan.org/fsw/ff/00index_e.html
<graphics.h>	Higher-level hardware-independent graphical primitives. It is assumed that a graphics mode is already initialised on a lower level of execution.
<platform.h>	Library with specific functions and definitions related to the exact hardware platform on which the C.impl interpreter is running.

3.3 The <platform.h> Library for C.impl

The library provides functions and definitions to work with the specific hardware platform, in this case – the ELLO 1A system.

Note: The functions in the <platform.h> library only apply to the ELLO 1A system.

3.3.1 System

void reset(void)

Immediately reset the system.

void delay_ms(unsigned long milliseconds)

Generate specified delay in milliseconds.

void set_timer(unsigned long milliseconds, void (*intHandler)())

Enable an internal timer counter which calls the specified function '*intHandler()*' every time when at least the given number of milliseconds have passed. The timer can be disabled by setting 0 in the '*milliseconds*' parameter.

C.impl allows multiple simultaneously working timers, each calling a different handler function.

3.3.2 Keyboard

void setKbdLayout(char country)

Set keyboard layout. This is a volatile setting and will restore back to the original code on a system reboot.

Currently supported keyboards codes are:

0 US International
1 UK Extended
2 DE (Germany, Austria)
3 FR (France, French-speaking countries)

char getKbdLayout(void)

Return the current keyboard layout.

void setBrkCode(char code)

Set ASCII code if the break key (normally 3 for Ctrl-C).

This is a volatile setting and will restore back to the original code on a system reboot.

char getBrkCode(void)

Return the current break code.

3.3.3 Sound

void beep(void)

Generate a 100 milliseconds long beep tone with frequency 945 Hz.

void sound(int freq, int vol)

Generate sound with specified frequency in Hertz and volume in the range between 0 and 1000.

The sound will continue until a new ***sound()*** function sets different parameters. It can be stopped completely by supplying 0 to either the frequency or the volume parameters, or both.

3.3.4 Video

These functions are the low-level layer for a video output. They ensure initialisation of a video mode, and read/write of a single pixel on the screen.

void initVideo(int mode)

Initialise video mode. ELLO 1A only supports video mode 0 (480x320, mono). Any value other than 0 in the parameter, may cause unpredicted behaviour in ELLO 1A.

int getVmode(void)

Return the current active video mode.

int Hres(void)
int Vres(void)

Return the video screen horizontal and vertical resolution in number of pixels, or -1 in case the screen is does not support graphics.

void clearScreen(int colour)

Fill the entire screen with a given colour.

int getPixel(int x, int y)
void setPixel(int x, int y, int c)

Read and return the colour of, or set a single pixel on the screen at coordinates (x,y) in colour (c).

3.3.5 I/O Ports

Ports are accessed as regular read/write operations in the memory map. The library contains predefined constants for the base address of each port register, as well as constants referring to offset from the base for registers related to the port.

Thus for example, writing 0xFF into the PIC32's LATB register will be expressed as ***(BASEB + LATA) = 0xFF;***

As another example, clearing the specific bit RA0 would be ***(BASEA + LATCLR) = BIT(0);***

Referring to invalid port numbers in C.impl may cause unexpected system behaviour.

3.3.6 Communications

int spiOpen(int channel, int mode, int baudrate)

Open a SPI channel for communication. Will return 0 is successful, or a negative value result otherwise.

ELLO 1A has only one SPI channel, which is shared among all SPI devices (including the SD card), so the only acceptable parameter for channel number, is 1.

The '*mode*' parameter specifies the SPI mode between 0 and 3. The port is always initialised for exchange of data words with length of 8 bits.

The '*baudrate*' parameter defines the speed of the transfer.

int spiClose(int channel)

Close a SPI channel. Will return 0 is successful, or a negative value result otherwise.

The only acceptable parameter for '*channel*' in ELLO 1A, is 1.

unsigned char spiByte(int channel, unsigned char data)

Perform data exchange of a single byte through an open SPI channel. A SPI transfer involves simultaneous transmit and receive operation on every transfer.

It is important to mention, that the chip select signal for the SPI device participating on the other end of the communication, must be set low prior to the actual transfer taking place, and set back high when the transfer is complete.

The only acceptable parameter for '*channel*' in ELLO 1A, is 1.

void spiBlock(int channel, unsigned char *buffer, size_t len)

Perform data exchange of data block of '*len*' bytes through an open SPI channel. The data to be transmitted should be preloaded in the buffer, and the received data is returned on its place.

It is important to mention, that the chip select signal for the SPI device participating on the other end of the communication, must be set low prior to the actual transfer taking place, and set back high when the transfer is complete.

The only acceptable parameter for '*channel*' in ELLO 1A, is 1.

3.4 The <graphics.h> Library for C.impl

void drawLine(int x1, int y1, int x2, int y2, int c)

Draw line from point (x1,y2) to point (x2,y2) and colour (c).

void drawFrame(int x1, int y1, int x2, int y2, int c)

Draw a rectangular frame built from four lines between points (x1,y1) and (x2,y2), and colour (c).

void drawRect(int x1, int y1, int x2, int y2, int c)

Draw a solid rectangle between points (x1,y1) and (x2,y2), and colour (c).

void drawTriangle(int x1, int y1, int x2, int y2, int x3, int y3, int c)

Draw a solid triangle between points (x1,y1), (x2,y2), and (x3,y3), and colour (c).

void drawCircle(int x, int y, int r, int c)

Draw a circle with centre point (x,y) and radius (r) and colour (c).

If the radius is specified as a positive number, the circle is drawn solid. If (r) is negative, only a circle frame is drawn.

void drawEllipse(int x, int y, int rx, int ry, double tiltAngle, int c)

Draw an ellipse with centre point (x,y), two radiuses (rx) and (ry), and colour (c). The parameter 'tiltAngle' specifies the tilt (rotation) angle of the ellipse in measure of radians.

If both radiuses are given as positive numbers, the ellipse is drawn solid.

If both radiuses are given as negative numbers, the ellipse is drawn as a frame.

If one of the radiuses is positive while the other is negative, nothing is drawn.

void drawSector(int x, int y, int rx, int ry, double tiltAngle, double startAngle, double endAngle, int c)

Draw a sector of ellipse, whose centre is at (x,y), radiuses (rx) and (ry), tilt angle (tiltAngle) in radians, and colour (c).

Two additional angular parameters (startAngle) and (endAngle), both in measure of radians, specify the actual draw sector as part of a full ellipse.

In case the radiuses (rx) and (ry) are equal, the ellipse is transformed into a circle.

If both radiuses are given as positive numbers, the ellipse is drawn solid. If both radiuses are given as negative numbers, the ellipse is drawn as a frame. If one of the radiuses is positive while the other is negative, nothing is drawn.

void floodFill(int x, int y, int c)

Flood fill with colour (c) an enclosed area. The start point (x,y) must be inside the area to be filled.

void getRect(void *buffer, int x1, int y1, int x2, int y2)

Get a rectangular area from the screen into a memory buffer. The buffer must be pre-allocated and with sufficient size. The minimum needed buffer size can be calculated with the formula:

$$Buffer_{(bytes)} = 8 + (H * V) / 8$$

Where H and V are the horizontal and vertical dimensions of the rectangle in pixels.

*void putRect(void *buffer, int x, int y, int opr)*

Restore a rectangular area from a buffer on the screen starting from specified coordinates of the top left corner of the area.

The last parameter defines the logic type of restoring the pixels:

OPR_COPY	0	All pixels from the buffer overwrite the pixels on the screen.
OPR_OR	1	Logical operation "OR" is performed between every pixel from the buffer and its destination on the screen.
OPR_XOR	2	Logical operation "Exclusive OR" is performed between every pixel from the buffer and its destination on the screen.
OPR_AND	3	Logical operation "AND" is performed between every pixel from the buffer and its destination on the screen.
OPR_NOT	4	All pixels are restored on the screen in their inverted form.

Unknown operation types are assumed as type 0. Constants "OPR_xxx" as per the table above are predefined in the library.

*void drawShape(int x, int y, double tiltAngle, char *def)*

Draw a vector-defined shape, starting from point (x,y) and with tilt angle (rotation) given in measure of radians.

The shape definition is a text string consisting of commands and vectors:

M [relX],[relY]	Move to a new point with relative coordinates (relX,relY)
C [col]	Set new drawing colour and enable draw when moving
D	Enable draw when moving
N	Disable draw when moving
F	Flood fill taking the current position as start point

Whitespaces are ignored in the definition string. Vectors and colours also allow use of hexadecimal numbers, preceded by '***X***' or '***x***' character. Missing numbers are assumed 0. Thus, a command "*M,10*" is a functional equivalent to "*M0,10*". Command "*M-10,*" is a functional equivalent to "*M-10,0*".

Predefined colour constants are not supported in the shape definition string. Colours must be specified in a numeric form.

An example for vectored shape:

drawShape(160, 130, 0.0, "C-3 M10,20 M20,10 M-20,10 M-10,20 C1 M-10,-20 M-20,-10 M20,-10 M10,-20 N M,15 C-12 M,30 N M-15,-15 D M30,");

This definition will draw a four-rayed star with green left part and red right part (on a system that supports colours), and a blue cross inside, starting from the top point at screen coordinates (160,130) and not rotated.

void drawChar(int ch)

Draw a character from the currently active font.

In scale factors greater than 2, the drawing function applies an algorithm to smooth down the blocky appearance of the large characters.

void lockScroll(void)

Prevent the screen from automatically scroll up when printing characters normally would require it to.

void unlockScroll(void)

Restore the automatically scroll up when printing characters requires.

void posX(int x)
int posX(void)

Set or return horizontal position for the next text character.

void posY(int x)
int posY(void)

Set or return vertical position for the next text character.

void fontScale(int factor)
int fontScale(void)

Set or return the scale factor for drawing font characters.

void fontFcol(int col)
int fontFcol(void)

Set or return the drawing colour for font characters.

void fontBcol(int col)
int fontBcol(void)

Set or return the background colour for font characters.

void fontSet(font_t *font)

Activate custom font. The parameter (*font) is a pointer to a font structure. Once activated, the font will be used by all following functions which output characters on the screen.

3.4.1 Fonts

C.impl supports drawing characters from different fonts. Font are defined as a sequence of individual bitmaps for each character, and characters with variable width within the same font are supported.

A font structure is defined as:

```
typedef struct font_t {
  font_header_t header;
  unsigned char definitions[];
};
```

A separate header structure holds the general parameters of the font:

```
typedef struct font_header_t {
  unsigned short start;          // code of the first character in the font
  unsigned short characters;     // number of character definitions in the font
  unsigned char width;           // font width
        // This parameter specifies the number of columns in the characters.
        // In fonts where the field (.width) is 0, every character definition starts
        // with a byte that defines how many columns are present in this character.
        // The actual number of bytes to follow depend on the height of the font as
        // well: for fonts with height 8 lines or less, every byte represents one
        // column, for fonts with height 16 lines or less, every column takes two
        // bytes, and so on.
        // In fonts with fixed width where (.width) is greater than 0, the leading
        // width-specifying byte in every definition is missing since the width is
```

```
        // already know for all characters
  unsigned char height;  // character definition height in pixels
        // This parameter also inherently defines the number of bytes needed for one
        // column of a character
  unsigned char blankL;// blank columns to add on the left side of every char
  unsigned char blankR;// blank columns to add on the right side of every char
  unsigned char blankT;// blank rows to add on the top side of every character
  unsigned char blankB;// blank rows to add on the bottom side of every char
  char *name;                    // optional font name as ASCIIZ string
};
```

Font definition examples:

1. The definition of character 'A' in a font with fixed width 5 (the "width" field in the font header has value 5 and that defines that all characters in the font will be within 5 columns) and height 7 rows:

   This sequence of bytes looks like this:

   0x7C, 0x12, 0x11, 0x12, 0x7C

	7C	**12**	**11**	**12**	**7C**
0			#		
1		#		#	
2	#				#
3	#				#
4	#	#	#	#	#
5	#				#
6	#				#
7	*not used*				

In addition to the definition, in the font header there are fixed definitions for certain number of blank pixels to be added to all sides of every character in the font.

2. Definition of character 'W' in a font with variable width (the "width" field in the font header has value 0) and height 14 rows.

Since the width is variable, every character in the font has its own width, and that is specified in one additional byte at the beginning of every character definition.

The character in this particular example has 12 columns, and every column is described by two bytes. Since by definition the font has maximum height of 14 pixels, the last two bits in the second byte will remain unused and they are not displayed on the screen.

The definition will look like this:

12, 0x01,0x00, 0xFE,0x01, 0x00,0x07, 0x00,0x20, 0x20,0x18, 0xE0,0x07, 0x20,0x1e, 0x00,0x20, 0x00,0x18, 0xFE,0x07, 0x01,0x00

	01	**FE**	**00**	**00**	**00**	**20**	**E0**	**20**	**00**	**00**	**FE**	**01**
0	#											#
1		#									#	
2		#									#	
3		#									#	
4		#									#	
5		#				#	#	#			#	
6		#					#				#	
7		#					#				#	

	00	**01**	**07**	**1E**	**20**	**18**	**07**	**1E**	**20**	**18**	**07**	**00**
8		#	#				#				#	
9			#	#			#	#			#	
10			#	#			#	#			#	
11				#		#		#		#		
12				#		#		#		#		
13					#				#			
14	*not used in this font*											
15												

All character definitions start immediately after the font header, and follow up in sequential order, so in the example above, after the character ‘W’ will follow character 'X', then character ‘Y’, and so on.

C.impl includes a standard system font with fixed size in a 5x8 pattern drawn in a 6x11 area. The font is based on the full 255 characters CP437 map with the only change introduced in code 158 to include the euro sign character.

3.4.2 Colours

C.impl operates natively with 24-bit colours, however in a monochrome graphic system like the ELLO 1A, any colour greater than 0 is the same. Simulated "colours" are however possible, by using specific graphical patterns. Those simulated colours are given to the interpreter as negative numbers.

The <platform.h> library for ELLO 1A includes many such predefined patters:

Colour constant	Numeric value	Pattern
COL_SOLID	0xFFFFFF	Main colour (solid white/active)
COL_NONE	0	No colour (solid black)
COL_INVERT	-1	Opposite to the current colour
COL_TRANSP	-2	No colour (transparent)
COL_GREY50	-3	50% grey
COL_GREY25	-4	25% grey
COL_HLN50	-5	50% horizontal lines
COL_HLN33	-6	33% horizontal lines
COL_HLN25	-7	25% horizontal lines
COL_VLN50	-8	50% vertical lines
COL_VLN33	-9	33% vertical lines
COL_VLN25	-10	25% vertical lines
Reserved	-11	
COL_DGL33	-12	33% / diagonal lines
COL_DGL25	-13	25% / diagonal lines
Reserved	-14	
COL_DGR33	-15	33% \ diagonal lines
COL_DGR25	-16	25% \ diagonal lines
COL_SPL33	-17	33% / dotted diagonal lines
COL_SPL25	-18	25% / dotted diagonal lines
COL_SPR33	-19	33% \ dotted diagonal lines
COL_SPR25	-20	25% \ dotted diagonal lines
Reserved	-21 … -29	
Percentage fill	-30 … -50	A pixel is turned on with specific probability.

		The value -30 is equivalent to probability 0 (pixel is never on). Every next value is increasing the probability by 5%. Thus, value -31 is probability 5%, value -32 is probability 10%, and so on. Value -50 represents probability 100% (pixel is always on).

4 Frequently Asked Questions

4.1.1 Why was PIC32 chosen for ELLO 1A?

Due to several reasons. First, it is a very popular and well known chip, an active product with large supporting code base and community of users. It is one with which I am acquainted quite well, and has proved itself in a number of other projects that I have made in the past. The programming tools for it are also easily available and don't cost much. The chip itself is very robust and tolerates mistakes such as shorted pins or incorrect voltages, it has all the essentials without being bloated with rarely used stuff, and most importantly – comes in a DIP package.

As far as my knowledge goes, at the moment of building the ELLO 1A there was no other microcontroller with similar or better characteristics than PIC32MX170/270, also available on the market in through-hole package.

I have been appealing for some time now to Microchip for a release of the PIC32MZ version in DIP package. If that happens some day in the future, there could well be a more powerful version of ELLO for it too.

4.1.2 Are there any known issues and limitations?

There are some limitations and deviations from the standard in C.impl. Most of those are down to the fact it is an interpreter.

1. Function pointers and function address dereferencing are not supported.

2. In for(), while(), and do...while() loops, 'break' and 'continue' can only work from within a {} block.

3. The 'goto' instruction only allows jumping to a location at the same or upper {} depth level.

4. A goto label must have unique name among all labels globally.

5. The 'extern' keyword is not supported since the entire program and included libraries are regarded as a single source file.

6. In the ternary operator ?: structure, there must be a space before the colon ':' so it is not mistaken for a label.

7. The main() function must not have parameters or a return value.

8. In composite data types (enum, struct, union), the name of the type must be before the {} block, but can't be after.

9. Structures, unions, and enums are all supported but have high runtime memory cost and should be used with care.

10. Keywords 'struct', 'union', and 'enum' must be used only for the initial type declaration, but not for declaring later instances.

11. Casting to structure or union pointer will not work. The cost of its implementation in an interpreter is massive in terms of both runtime memory and execution speed.

12. Indexing by pointers (eg. int a[10]; int b = *(a+1);) will not work for data types other than 8-bit. Indexing needs to be done in the brackets way.

4.1.3 What is the license for this project?

ELLO is an open-hardware project. There are no limitations for its use in any way the user sees fit.

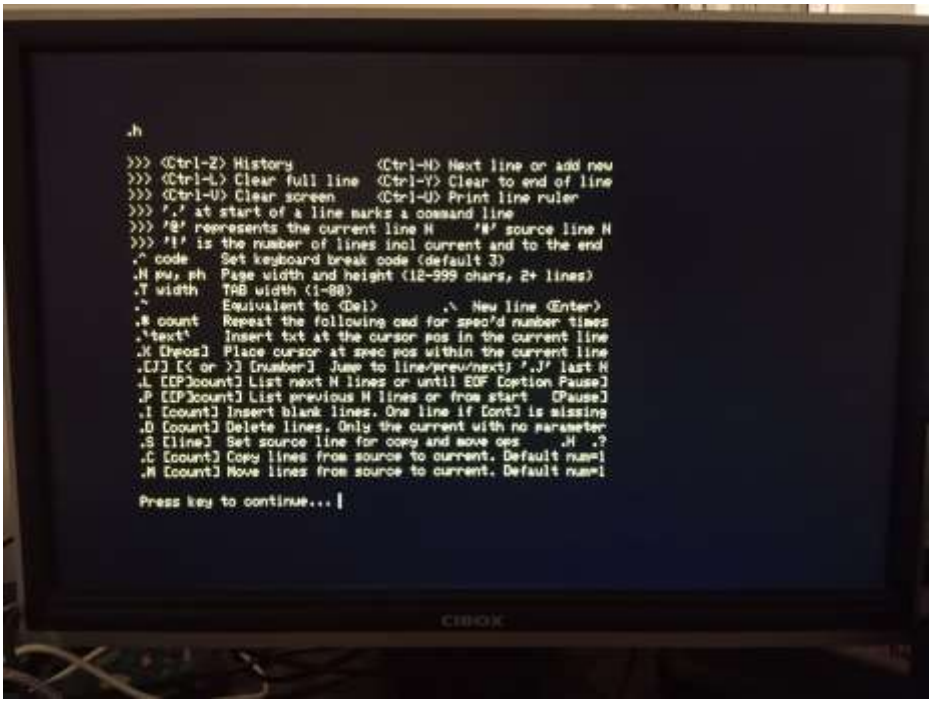
The source codes for its software are currently available by request to contributors only, but my plan is to have it fully released as open-source, once it reaches a more mature and tested stage.

If you want to become a contributor, join and write in the FB group: https://www.facebook.com/groups/ellocc

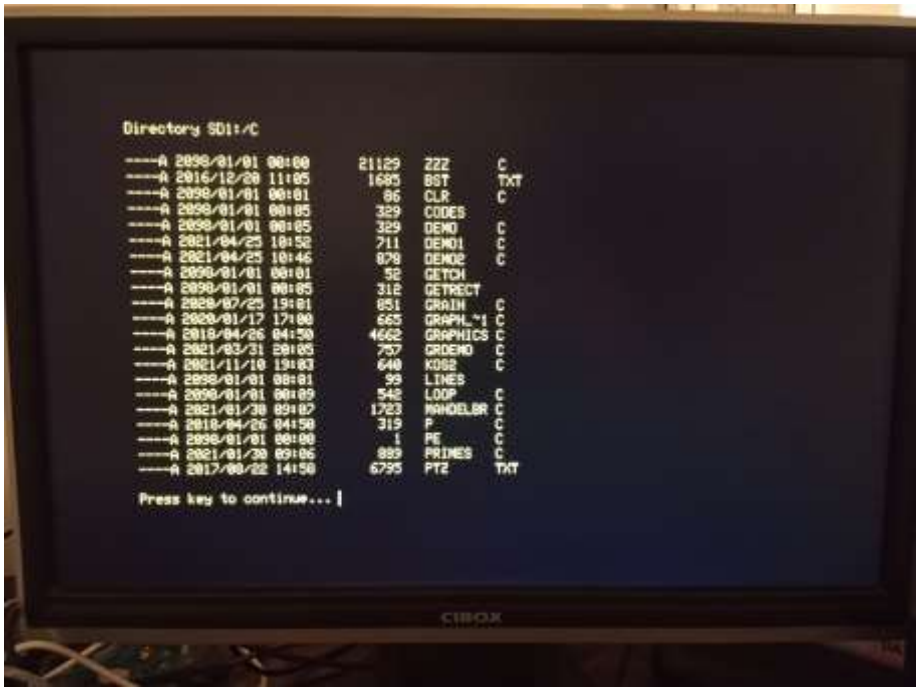
5 Screenshots

The login screen

After login

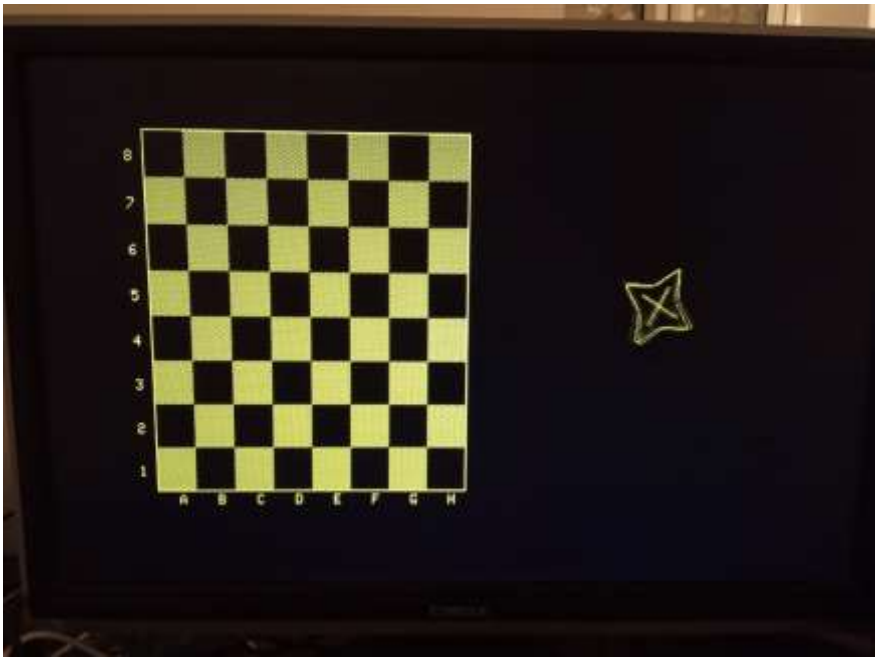


The help page



Listing the files on SD1:

Writing code in the text editor



Running the code (the blurred part is moving)

6 Downloads

You can buy boards for ELLO 1A from **my Tindie store**

(PDF) Schematic

(XLS) BOM for Through-Hole assembly

(XLS) BOM for Surface-Mount assembly

(PDF) Assembly Drawing

(ZIP) PCB Gerbers pack

(ZIP) Full manufacturing pack

(ZIP) Firmware for PIC32MX170 or PIC32MX270

> ***NOTE:*** *The firmware is developed on PIC32MX270 in order to ensure future support for USB functionality. When programming a PIC32MX170, the programmer will display a warning that the device is not matching the intended target.*
>
> *Ignore the warning and upload the code into the microcontroller.*

Facebook group: https://www.facebook.com/groups/ellocc

7 Change Log

R120 07/03/2022

Graphics changed from 480x320 to 480x250 to enable more available system memory.

Added ruler in RIDE (Ctrl-U).

Multiple corrections and improvements.

R114 15/06/2021

Introduced internal library callback functions for support of hardware-driven events.

The <platform.h> library enriched with functions for work with the serial port expansion in ELLO 1A.

RIDE and C.impl ported on a PIC32MZ chip.

Multiple improvements and bug fixes throughout the project.

R113 21/05/2021

Updated dynamic memory manager library for more efficient work in situations when the available free memory is extremely low.

R112 24/04/2021

Introduced support for the I2C interface (single master mode only). Added I2C functions to the <platform.h> library.

ELLO 1A now supports an optional real-time clock/calendar of a type compatible with DS3231, connected on the I2C bus. Added RIDE commands for setting time and date.

Improved ./lsl command in RIDE with additional information about the C.impl libraries.

The FatFs library upgraded to its latest version 0.14b.

Introduced support for entering international characters with non-US/UK keyboards as well as combinations with dead keys.

R111 13/04/2021

Main focus in the release put on correction of multiple issues in C.impl.

PS/2 keyboard functions revamped to provide support for input of extended character set. Some characters in the built-in font were also edited for better appearance and clarity.

Optimised country layout codes.

Expanded settings structure to include validity checksum, and tab width and console page size. Settings from previous versions will revert to default after first run.

R110 03/04/2021

Graphic resolution reverted back to 480x320 pixels with a view to ensure support for a future LCD-based mobile version. Monitors set for a previous version of the software, may need readjustment.

Text resolution changed from 60x27 characters, to 80x29 characters.

Default system font footprint changed from 8x12 pixels to 6x11 pixels.

Source file tree reorganised and optimised with multiple small improvements.

Support for #pragma options removed and replaced by new functions and new RIDE commands.

Keyboard layout and break code set by environment commands have permanent effect; the corresponding functions in <platform.h> stick to volatile model.

Fixed issues with memory allocation in copy/move operations in RIDE.

Errata note for rL2 PCB:

The component model for OSC1 should be read as *MXO45HSLV-3C-20M000000*

R102 26/03/2021

Fixed bug in getRect() to prevent the screen area being cleared during the process.

Startup process first checks for existence of “IFS:/autorun” and if one is not found, checks for an alternative in “SD1:/autorun”.

Introduced support for USB console *(in systems with PIC32MX270 only)*.

R101 18/03/2021

Graphic functions getRect() and putRect() and function constants.

Improved operation in console mode.

R100 16/03/2021

First “final” release.

Added support for serial console with protocol 38400/8N1.

Presence of connected video monitor is tested and the processor freed from video-supporting tasks in case a monitor is not detected.

Percentage fill colours.

R1c 06/03/2021

Candidate release.

Default video resolution (video mode 0) changed from 480x320 to 480x324 for better utilisation of RAM and accurate text alignment on screen.

R1b 28/02/2021

Beta release.

Hardware files and schematic updated to revert back to signal output voltage clipping. Added new components R4 and D1 in the BOM.

Added support for SPI and PIC32’s I/O ports.

Introduced timed interrupts.

R1a 25/02/2021

Initial alpha release.