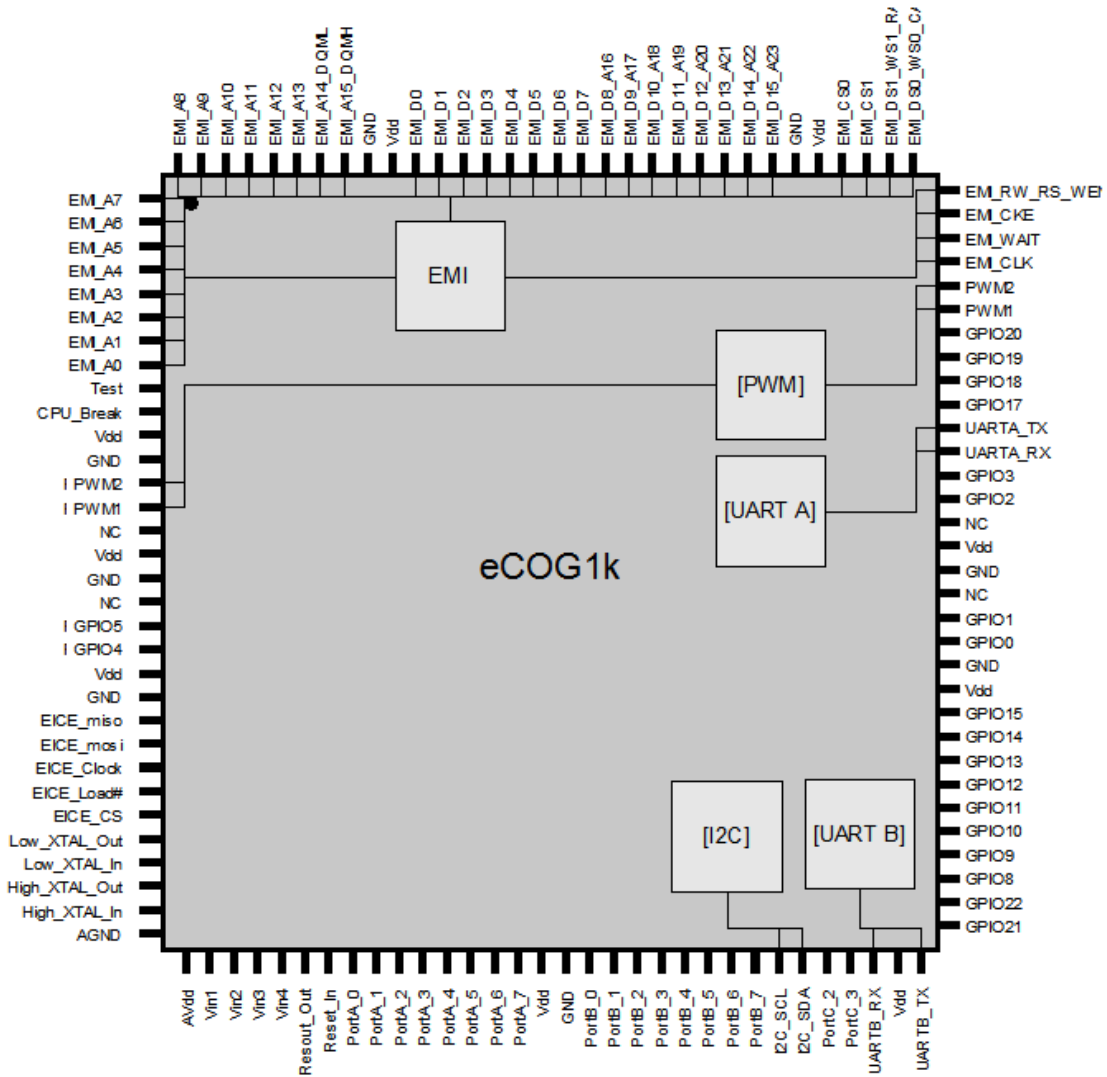


# AN020 – Simple Command Line Interface

## Version 1.1

This application note describes a simple Command Line Interface for use with the eCOG1k microcontroller.



## Confidential and Proprietary Information

©Cyan Technology Ltd, 2008

This document contains confidential and proprietary information of Cyan Technology Ltd and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.



## Revision History

Version	Date	Notes
V1.0	08/12/2004	Initial Release
V1.1	11/03/2005	Updated for CyanIDE V1.1

## Contents

1	Introduction .....	5
2	Glossary .....	5
3	Command Line Interface Description .....	6
3.1	Overview.....	6
3.2	Syntax.....	6
3.3	Command Naming Conventions.....	6
4	Implementation .....	7
4.1	Adding New Commands.....	7
4.2	Parameter Passing and Errors .....	8
4.3	Evaluation Board Examples .....	8
5	Predefined Commands.....	8
5.1	Arithmetic.....	8
5.2	Stack Manipulation .....	8
5.3	Characters and Strings.....	9
5.4	Variables and Constants .....	9
5.5	Defining Words and Flow Control.....	10
Appendix A	LCD Implementation .....	11
A.1	Overview.....	11
A.2	New LCD Commands.....	11
A.3	Implementation of LCD." Command.....	11

## 1 Introduction

This application note describes a simple Command Line Interface for use with the eCOG1k microcontroller.

The Command Line Interface provides a simple method for initial testing and experimentation with new hardware. The user communicates with the Interface through one of the Serial Ports on eCOG1k, and the syntax of the Interface is based on the FORTH programming language.

The minimum hardware requirement for the Interface is just a working eCOG1k and an RS232 level translator for the appropriate serial interface.

## 2 Glossary

A table of abbreviations used in this document.

ADC	Analogue to Digital Converter
CLI	Command Line Interface
eCOG1	Cyan Technology target micro controller
LCD	Liquid Crystal Display
RPN	Reverse Polish Notation
UART	Universal Asynchronous Receiver/Transmitter

## 3 Command Line Interface Description

### 3.1 Overview

The CLI is meant to provide a simple interface for the testing and debugging of the users hardware. It offers the following advantages:

- Written in C and built using the CyanIDE tool chain, available free from Cyan Technology Ltd.
- Readily and simply extendable to include new commands for the target application.
- A defined syntax and parameter handling.
- Fully programmable in its own right, with conditional execution and looping for generation of more complicated testing functions.

### 3.2 Syntax

The syntax of the command line interface is based on that of the Forth programming language. Forth is a stack based system and so uses Reverse Polish Notation, as used by some H-P calculators.

Characters are received from the serial interface and are stored in an input buffer until a Ctrl-M (Enter) character is detected. Then the contents of the buffer are parsed and each command in the buffer is processed. If an error occurs, the rest of the buffer is thrown away and the CLI returns to waiting for a new input line.

A command is defined as a group of non-whitespace characters delimited by whitespace. Commands can be placed sequentially on a line up to the maximum size of the input buffer.

All the commands that are understood by the CLI are kept in a dictionary. When a command is entered, the dictionary is searched, and if a matching word is found in the dictionary, then the function associated with that word is executed.

If the word is not found in the dictionary, then it is tested to check if it is a number. Any numbers received in the input buffer are placed on the top of the stack. The stack is a First-In-Last-Out buffer that is used to contain any parameters for the words that are executed.

If the entire input buffer is parsed without error, then the message "ok" followed by a Carriage Return and Line Feed is displayed. This applies even if there were no commands in the input buffer.

The parameters for a command are taken from the stack, and any results are placed back on the stack.

The "." (pronounced "dot") command word is used to remove the top item from the stack and display it. Thus if the command string "1 ." (without the quotes) is entered, the CLI responds with "1 ok". The CLI looks for a command "1" in the dictionary and does not find it, checks that it is a number and places it on the stack, then the "." command removes it from the stack and prints it.

### 3.3 Command Naming Conventions

The following rules apply to the names of commands.

- The maximum length of a command name is 32 characters.
- A command can be made up of any upper case letters, numbers and symbols. The CLI converts all letters to upper case before searching the dictionary.

The following conventions are used in the naming of commands. These are only suggestions.

- Commands that print anything are preceded with a full-stop.  
For example, the command to print all the known commands is `. WORDS`
- Commands that fetch a value and leave it on the stack finish with the @ symbol.
- Commands that store a value taken from the stack finish with the ! symbol.

## 4 Implementation

The implementation consists of three source code files: *kernel.c*, *memory.c* and *rs232.c*, and their associated header files.

The file *kernel.c* contains all the core functionality for the CLI, for parsing the command line and searching the dictionary. It also contains the C definitions for the predefined commands.

*Memory.c* includes the structure containing of all the predefined words, and the definitions for the array of integers that is used as the memory for the CLI command.

*RS232.c* contains the serial interface control used by the CLI. It contains the hardware-specific definitions of *putchar()* and *\_getchar()* that are used by the Standard I/O library.

The rest of the project is the start up code, interrupt vectors and main function. The main function initialises the CLI and then runs an infinite loop that executes two functions, to wait for the command line to be entered and then to parse the command line.

```
int main(int argc, char* argv[])
{
    RS232_Initialise();
    Coldstart();
    printf("\r\nCyan eCOG1 CLI\r\n");
    while(TRUE)
    {
        Expect();
        Process_TIB();
        OK();
    }

    return 0;
}
```

### 4.1 Adding New Commands

Adding new built-in commands to the CLI is quite straightforward.

First, write the code and wrap it in a function that does not take or return any parameters. For example:

```
void test( void )
```

Then add the function to the dictionary structure in the file *memory.c*.

The structure contains three values for each entry:

- The length of the command name.
- A pointer to a constant character array containing the command name.
- A pointer to the function to be called when the command is entered.

Thus to add the `test()` function above, the following line is added to the dictionary definition:

```
{4, "TEST", test},
```

When the project is built and loaded into the eCOG1k, the new command is available.

## 4.2 Parameter Passing and Errors

As the functions added to the CLI cannot directly accept or return parameters, any values required by the functions must be passed on the stack.

Two built-in functions are used to manipulate the stack: Push and Pop.

Push accepts a single integer value and places it on the top of the stack.

Pop removes the value from the top of the stack and returns it.

If an error occurs with either of these commands, for example if the stack is full when the Push function is executed or the stack is empty when the Pop function is executed, a global error flag is set that can be used to test for faults. This flag is called `Code_Fault_Flag` and can be checked or set by the function. If it is set when execution returns to the CLI parser, an error is assumed to have occurred in the execution and the rest of the input buffer is discarded.

## 4.3 Evaluation Board Examples

Several examples of commands are given in the file `devboard.c`. These can be used to control devices on the eCOG1k Development Board, such as the LEDs, switches, sounder and ADC. A more detailed description of the implementation of the LCD functions is given in Appendix A.

## 5 Predefined Commands

The descriptions in this section have the following format:

```
<Command Name>      ( <Initial Stack> -- <Final Stack> )
```

```
<Description>
```

The Command Name is the exact entry in the CLI dictionary.

The two stack states are the values on the stack before and after the command is executed. The top of the stack is to the right in both cases.

The description describes the function in words.

### 5.1 Arithmetic

```
+      ( n2 n1 -- n2+n1 )
```

This command adds the first and second values on the stack, leaving the result on the stack.

```
-      ( n2 n1 -- n2-n1 )
```

This command subtracts the first value on the stack from the second value on the stack, leaving the result on the stack.

```
*      ( n2 n1 -- n2*n1 )
```

This command multiplies the first and second values on the stack, leaving the result on the stack.

```
/      ( n2 n1 -- n2/n1 )
```

This command divides the second value on the stack by the first value on the stack, leaving the result on the stack.

### 5.2 Stack Manipulation

```
DROP  ( n -- )
```

This command removes the first value from the stack and discards it.

```
.      ( n -- )
```

This command removes the first value from the stack and displays it.

DUP ( n -- n n )

This command duplicates the first value on the stack, leaving both values on the stack.

SWAP ( n2 n1 -- n1 n2 )

This command swaps the order of the first and second values on the stack, leaving both values on the stack.

OVER ( n2 n1 -- n2 n1 n2 )

This command duplicates the second value on the stack, leaving the copy on the top of the stack.

### 5.3 Characters and Strings

CRLF ( -- )

This command prints the Carriage Return (Control-M) and Line Feed (Control-J) characters to the RS232.

EMIT ( n -- )

This command prints the character represented by the value on the top of the stack to the serial port.

." ( -- )

This command (pronounced "Dot-quote") will print the string immediately following it to the serial port. It continues until it reaches either the end of the line or a double quote (") in the line. For example: ." Hello" displays the text "Hello". Note that the first space between the ." and the string is not printed as it has to be there to separate the commands.

### 5.4 Variables and Constants

CONSTANT <NAME> ( n -- )

This command creates a new command of the given name that puts a given value on to the stack when it is executed.

For example: 27 CONSTANT gain creates a new command GAIN that puts the value 27 on to the top of the stack when it is executed.

VARIABLE <NAME> ( -- )

This command creates a new command of the given name and allocates a word of Forth memory for it. When the command is executed, the address of the memory location is placed on the stack.

For example: VARIABLE test creates a new command TEST that leaves the address of its associated Forth memory location on the stack when it is executed.

@ ( a -- n )

This command (pronounced "Fetch") fetches the value from the Forth memory address specified by the first value on the stack, and leaves it on the stack.

For example: using the variable TEST defined above, TEST @ fetches the value of the variable TEST on to the stack.

! ( n a -- )

This command (pronounced "Store") stores the second value on the stack at the Forth memory address given by the first value on the stack.

For example: using the variable TEST defined above, 12 TEST ! stores the value 12 in the variable TEST.

? ( a -- )

This command (pronounced "Query") fetches the value from the Forth memory address given on the top of the stack and prints it. It is effectively a combination of the @ and . commands.

## 5.5 Defining Words and Flow Control

It is not the intention of this document to describe all the programming details of the FORTH language, but a quick overview is given here. For more information, see the Forth Interest Group web site at <<http://www.forth.org>>.

To define a new command in Forth, start with a colon (:), a space and the name of the new command. The colon has the effect of putting the CLI into compile mode, and any command now entered is compiled into the new word rather than being executed. To finish the definition, use the semicolon command (;). The definition can span as many lines as needed for clarity and the OK prompt is only printed when the semicolon has been executed to define the end of the command.

As an example, a new command called `INC` could be defined to increment the value on the top of the stack. To do this it must place a `1` on the stack and then use the `+` command to add it to the original value on the top of the stack thus:

```
: INC 1 + ;
```

Within a new command, a range of flow control and conditional execution words can be used.

```
DO ( n2 n1 -- )
```

This command marks the start of a loop (terminated with the `LOOP` command). The two values on the stack represent the start and limit values of the loop counter, where `n1` is the start value and `n2` is one more than the end value. Within a `DO ... LOOP` structure, the command `I` can be used to place the current loop count on the top of the stack. So to print the value 1 to 9, the following code can be used:

```
: COUNT 10 1 DO I . LOOP ;
```

When the command `COUNT` is executed, the following numbers are printed:

```
1 2 3 4 5 6 7 8 9
```

Flow control decisions can be made using the `IF ... ELSE ... ENDIF` structure thus:

```
: TEST
  IF
    ." Non Zero"
  ELSE
    ." Zero"
  ENDIF ;
```

This command removes the top item from the stack, tests its value and prints the message `Zero` or `Non Zero` as appropriate.

```
0 TEST Zero ok
```

```
8 TEST Non Zero ok
```

Several built in comparison command words are available:

```
< ( n2 n1 -- T|F )
```

This command compares the top two values on the stack. If the second value is less than the first value, a non-zero (-1) value to represent TRUE is left on the stack. Otherwise, zero (FALSE) is left on the stack.

```
> ( n2 n1 -- T|F )
```

This command compares the top two values on the stack. If the second value is greater than the first value, a non-zero (-1) value is left on the stack. Otherwise, zero is left on the stack.

```
= ( n2 n1 -- T|F )
```

This command compares the top two values on the stack. If the second value is equal to the first value, a non zero (-1) value is left on the stack. Otherwise, zero is left on the stack.

## Appendix A LCD Implementation

### A.1 Overview

The implementation of some of the functions to control the LCD from the CLI is a little more complicated due to the string handling required.

### A.2 New LCD Commands

This appendix covers the implementation of five new words:

`LCD_INIT` ( -- )

This command initialises the LCD on the evaluation board using the driver library function `lcd_rst()`, and displays an initial string "Cyan eGOG1 CLI".

`LCD_CRLF` ( -- )

This command moves the current cursor position to the start of the next line on the LCD.

`.LCD` ( n -- )

This command removes the first value from the stack and displays it as a signed decimal number on the LCD.

`LCD_EMIT` ( n -- )

This command displays the character represented by the value on the top of the stack to the LCD.

`LCD."` ( -- )

This command displays the string immediately following it on the LCD. It continues until it reaches either the end of the line or a double quote (") in the line.

### A.3 Implementation of LCD." Command

The `LCD."` command is the most complicated, as it has to behave in different ways depending on whether it is being directly executed from the CLI or it is being compiled into a new command.

The first thing to do is to force the command to be executed immediately, regardless of whether or not the CLI is compiling a new word. To do this, a flag is set in the length field of the dictionary for the new command. The dictionary entry for the command is this:

```
{5 + IMMEDIATE_MASK, "LCD.\\"", LCD_Dot_Quote},
```

This causes the CLI to execute the command immediately, even if the CLI is currently compiling a new command.

The function can then decide what to do depending on the current state. The compilation state is reflected in a flag contained in Forth memory.

If the CLI is not compiling, the characters can be echoed directly to the LCD. Otherwise, the string has to be stored in the word currently being defined.

A run time version of the function that displays the string on the LCD is added to the current word. This run time function is in the dictionary, but is hidden by setting its length to zero. It is referenced by its position in the dictionary.