

EZ-Red – I/O power module

Programming manual

Find other manuals at <http://www.xonelectronics.it>

Index

Introduction.....	3
Hardware summary.....	3
Internal PLC applications.....	3
TSmon compiler / IDE.....	3
EZ-Red programming language.....	4
Introduction.....	4
Resources and data types.....	4
DEFINE and DECLARE.....	4
Multitasking.....	5
I/O: synchronous (default) or asynchronous.....	6
Program structure.....	7
Program format.....	8
Comments.....	8
Identifiers.....	8
Numbers (numeric literals).....	8
Labels (declaration).....	9
DEFINE (declaration).....	9
DECLARE (declaration).....	9
Instructions (statements).....	10
Assignments.....	10
Expressions.....	10
Operators.....	11
NOT (operator / function).....	11
Special operators (modifiers) for bits: /, \, ^, !.....	12
WAIT instruction.....	12
WAITREMAIN (residual time after a WAIT).....	13
Main resources (inputs, outputs, general hardware).....	14
Digital inputs X1..X8.....	14
XINVERT (logic inversion of inputs).....	15
XTHRESHOLD UP/DN (noise / anti bounce filter).....	15
XCOUNT1 and XCOUNT2 (rising edge counters).....	15
Power outputs Y1..Y8.....	15
YINVERT.....	15
Feedback (outputs failure alarm) FB12, FB34, FB56, FB78, FBBYTE.....	15
TIMERS: TIMERMS, TIMERSEC, TIMERMIN.....	16
FX inputs, FXCOUNT and encoder interface.....	16
FXCOUNTLx and FXCOUNTHx.....	16
Encoder.....	17
Encoder preset.....	17
Flow control (IF-THEN-ELSE and GOTO).....	18
GOTO (unconditional jump).....	19
Special instructions and resources.....	20
Task control.....	20
WAKEUP (task activation).....	20
RESTART (start again from the beginning).....	20
SUSPEND.....	20
CYCLERUN.....	21

Watch-dog.....	21
WDTFIRED e WDTSTOPSCYCLE.....	21
WDTOUTS, WDTAOUT1-2 (watch-dog outputs pattern).....	22
WDTTIME (PC communication time-out).....	22
WDTFBENA12-34-56-78 e WDTFBBYTE (power outputs monitoring).....	22
Interaction between watch-dog, PC and PLC cycle.....	22
LOG instruction (data acquisition and debug).....	23
REPORTBACK, SENDTOPC (inputs status transmission).....	23
Special functions for configuration, protection and diagnostics.....	23
CONFIGxxx (BIT, CHR, WRD).....	23
BCW_ xxx (BIT, CHR, WRD).....	24
DISABLEUSB.....	24
FIRSTRUN.....	24
FORCEDXS.....	24
PCCONNECTED.....	25
PWDPROTECT.....	25
Example programs.....	26
Press with safety pushbuttons (procedural).....	26
Slide with button and two limit switches (functional).....	26
Slide with button and two limit switches (procedural).....	27
Slide, procedural, using two tasks.....	28
Motor with encoder control.....	29

Introduction

Hardware summary

The EZ-Red is an interface module for PC, connected via USB, suitable for process control, automation and data acquisition at 24 volts. The module features:

- 8 digital inputs, and 2 fast and coupled, suitable for an optional quadrature encoder
- 8 power outputs at 24 volts nominal (from 12 to 30 volts)
- 2 analog inputs 0-10V, and 2 analog outputs 0-10V
- Internal programmable watch-dog, to monitor PC communications, power supply and outputs
- Flash memory (non volatile) to store PLC cycle and configuration variables

Internal PLC applications

EZ-Red can control external devices without a connection to a PC, in stand-alone mode. To do this, it is necessary to write a *PLC cycle*, or program, which is executed by the PLC. While the PLC runs, the module can be left alone, but it is possible to use a computer to interact with it; when the PLC is running, there are three possible scenarios:

1. Completely autonomous execution. Suitable for cases where a graphic user interface is not required, nor are mass memory, complex calculations, network access and so on. All the machinery can be managed by EZ-Red alone.
2. PLC cycle combined with a computer. The main task is executed by the module, while the PC only reads data from time to time, in order to implement a user interface or read data to log and store.
3. PLC cycle combined with a PC, where the EZ-Red implements parts of the process (perhaps the most time-critical ones) and the PC does the main work. The module is a parallel task or subroutine of a more articulated system.

TSmon compiler / IDE

The PLC cycle must be written with a text editor or the internal text editor of TSmon, the utility program supplied together with EZ-Red. This application, TSMON.EXE, is a true integrated development environment which can be used to write, modify, compile, debug, send cycles to EZ-Red and store them in non-volatile memory. Please refer to the “TSmon user manual” for details.

EZ-Red programming language

Introduction

The language has notations to read inputs, set outputs, store intermediate values, do calculations, control the flow of execution; it is a true programming language, similar in several aspects to Basic, mainly because it is *procedural* (commands are executed in the order they are written) and *imperative* (actions are described by imperative verbs like GOTO, WAIT, SUSPEND). Inputs and outputs are managed like variables which can be set and used in expressions, for example "Y1=ON" activates the digital output 1, while a command like "Y1=X1" copies the digital input 1 to the digital output 1 – a single statement contains a read and a write.

A common computer language, though, would not be very suitable for the EZ-Red because the target application requires easiness of I/O and timing control. So, the choice has been a hybrid language which inherits both from classic languages and PLC ladder, and the two philosophies can be used together. From ladder comes the concept that the hardware is seen as a set of *resources*, each with a unique name, like Y1 (the first power output), AOUT2 (the second analog output), TIMERMS1 (a timer which counts milliseconds); these resources are manipulated like variables, but with some extensions borrowed from the PLC world, like the notation for signal fronts.

Resources and data types

Nearly all programming languages have the concept of “data type”, and some of them also allow to create new ones. EZ-Red has only three types: unsigned 16 bit integers (word), its subtype made of 8 bits (byte), and the boolean type also called BIT which can only be on/true or off/false.

Resources are inputs, outputs, internal memories, timers, optional flags – well, all the hardware of EZ-Red the PLC cycle can manipulate. Every resource has its type: for example Y1 is of type BIT, while AIN1 (the first analog input) has type BYTE (unsigned 8 bit integer). Resources are read by specifying their name in an *expression*, and are written with an assignment. So, the statement “Y1 = X1” makes a read from X1 (first digital input, bit) and assigns that value to Y1 (first digital output, bit). The part to the left of the equal sign must be the name of a resource; the part to the right must be an *expression* of a compatible type. The most important resources, apart from inputs and outputs, are memories “R” (internal relays, bit) and “DT” (normal integer variables, word). “R”s and “DT”s are used to store temporary values and to compute numbers. There are more resources, for example timers, counters, special flags and so on.

Refer to the section *Expressions* for more informations.

DEFINE and DECLARE

The keyword DEFINE is used to assign a significant name to a resource or a constant, in order to make the program more readable and understandable. By using for example "DEFINE MOTOR Y1", it is possible to refer to Y1 using the name of the function that output performs; so it becomes possible and more clear to write "MOTOR=ON" and "MOTOR=OFF" to turn on or off the motor connected to the power output 1. By the way, even “ON” and “OFF” are internally implemented with a DEFINE. DEFINE is similar to the C language “#define”, but more restrictive.

If the target resource of a DEFINE is an input or an output, it is possible to declare it negated by using an exclamation mark (“!”). For example:

```
DEFINE stop !X2
```

creates an identifier "STOP" that refers to X2, but negated. When the digital input 2 is on (it has voltage), the identifier STOP is OFF (and the identifier X2 too). This notation makes it possible to “set it right” peripherals that have a negated logic. For example, a stop switch is often a normally closed contact and, when connected to an input, in “normal” condition would yield ON. It would be not very elegant to call it STOP, because the working cycle should run when “STOP=ON”. A negation in the DEFINE solves this.

Negated DEFINES work by setting other resources named XINVERT and YINVERT in the initial part of the

code generated by the compiler.

The keyword DECLARE is similar to DEFINE, but it is used to create variables instead of rename resources. DECLARE also wants an equal sign followed by an expression. The complete syntax is:

```
DECLARE [LOG] [DT|R] identifier = expression
```

The "R" or "DT" modifiers are not normally necessary: the data type is inferred from the type of expression; they are useful only in case of a recursive declaration like this:

```
DECLARE R GORUN = GORUN OR START
```

The "R" modifier is necessary because the expression contains a term (GORUN) which is not yet defined, so the expression can not be compiled.

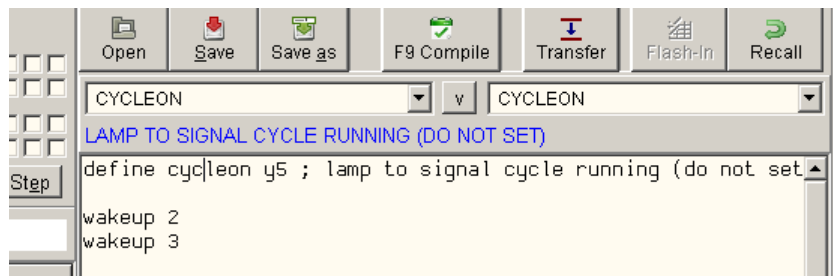
The modifier "LOG" specifies that writes (assignments) to the variable will cause an “event”, which will be notified to the connected PC (see LOG instruction for details).

In both cases of DEFINE and DECLARE, the specified identifier must be new, it is not possible to redefine existing ones. The keyword DEFINE, moreover, requires an existing resource to be referenced as target.

DEFINES and DECLAREs can have a comment on the same line after them, or the line just above; this comment is then associated to the identifier, to better explain it.

```
define motor y3      ; this comment will be the help message for MOTOR
                    ; this comment will be associated to CYCLECOUNT
declare CYCLECOUNT = 0
```

The TSmon utility uses this comment to show a help message when F1 key is pressed over an identifier:



Multitasking

The procedural model, compared to Ladder, is often more understandable because the attention can be focused on a single instruction, assuming that no other instruction interfere. But this advantage carries also the exact contrary: it is difficult in a procedural model make two things at the same time. For example, it is very easy to make an output pulse with a frequency of 1 hertz:

```
repeat:
  y1=on
  wait 500          ; <--- delay for 500 ms
  y1=off
  wait 500          ; <--- delay for 500 ms
  goto repeat
```

but it is much more complicated to make two outputs pulse with two different frequencies. The problem can be easily solved by using *tasks*, pieces of code executed together but independently from each other. The problem of

two different frequencies can be solved by using two times the code seen above, without employing timers or other hardware:

```

Task1:
  y1=on
  wait 500
  y1=off
  wait 500

Task2:
  y2=on
  wait 300
  y2=off
  wait 300
    
```

The content of task 1 terminates where task 2 begins. By definition tasks are repetitive – when there are no more instructions to execute they jump back to the first. The code fragment above is simple: task 1 turns on and off the output Y1, with 500 ms delays at Every step. The result is a square wave at 1 hertz. Task2 is about the same. EZ-Red can use up to 16 concurrent tasks, which give some benefits of the Ladder programming without having to totally embrace its philosophy. Tasks can also be suspended and restarted, so they can be used to build virtual hardware in software.

I/O: synchronous (default) or asynchronous

Classic PLCs update their I/Os once a cycle in an atomic way, thanks to an I/O copy in memory. This is normally effective, but sometimes it can lead to problems. This can be problematic even with an EZ-Red when used strictly in procedural mode. The standard way to update I/Os is when Task1, the first task, jumps backward (this comprises the implicit back jump at the end of the task) or when Task1 executes a WAIT instruction. Tasks other than Task1 must use a special syntax if they want to update immediately a single output, or they can force an I/O update. In the example above, actually, Task2 should use the following syntax:

```

Task2:
  y2<=on
  wait 300
  y2<=off
  wait 300
    
```

The special syntax is "<=" instead of a simple "=", and makes an immediate update of the specified output. Alternatively it is possible to invoke the special commands UPDATEX/UPDATEY/UPDATEXY. Another possibility is to set AUTOUPDATEXY to TRUE; this way, inputs and outputs are all updated immediately, step after step of any task.

If AUTOUPDATEXY is set (normally is OFF), a potential problem can happen in sequences like this:

```

; move motor forward or backward
mot_forward = goforward
mot_backward = NOT goforward
    
```

In this fragment two outputs, *mot_forward* and *mot_backward*, control a motor in the respective directions; it is not desirable that both the outputs are ON together. When AUTOUPDATEXY is set, though, this can happen: if in a previous cycle GOFORWARD was off, so MOT_BACKWARD is ON, when GOFORWARD becomes ON

MOT_FORWARD turns on before MOT_BACKWARD turns off. This is easily corrected by surrounding the critical instructions as the following code shows:

```

; move motor forward or backward
AUTOUPDATEXY = FALSE      ; disable temporarily
mot_forward = goforward
mot_backward = NOT goforward
AUTOUPDATEXY = TRUE       ; enable again
    
```

This trick should look familiar to those who know *interrupts*: synchronous I/O is disabled for a short time, then re enabled; this makes sure that no unwanted update can happen.

All in all, there are basically two different strategies to choose from:

- a) Leave AUTOUPDATEXY off (default), and use Task1 as a continuous cycle with no WAITs, like a normal PLC. The other tasks can rely on the updates made by Task1, if fast enough, or set directly the outputs with the syntax "Y1<=ON". WAIT instructions always read the true inputs, not the memory copy.
- b) Set AUTOUPDATEXY to ON and keep in mind that inputs and outputs are always updated, without a copy in memory. If there are critical sections, they can be protected in the way shown above.

If an output is read, the result is always the last value written, even if the physical output has not been yet updated. The following fragment, for example, has no effect at all on output Y1:

```

autoupdateXY = false
Y1 = on          ; Y1 is ON, but still there is no voltage on physical output
Y1 = not Y1     ; reading Y1 yields ON, so Y1 is turned off
    
```

Program structure

A PLC cycle (or program) is made of an INIT block of instructions acting as initialization, followed zero or more tasks. Every task starts with a label in the form TASKx, where x ranges from 1 to 16, and ends when the next task begins (with another label) or the program text is ended. Both INIT section and tasks are optional. If tasks are declared, the first must be Task1 and the following ones must be consecutive. The INIT part is executed only once, while the tasks are iterative.

```

; the INIT part (optional), precedes any other "TaskX:"
y1=x1          ; executed once at start-up
wait 1000      ; executed once

Task1:         ; beginning of the first task
...           ; instructions executed repeatedly
; an implicit "GOTO Task1" is put here by the compiler
    
```

If no labels like "TaskX:" are present, the program consists of only the INIT part and will be executed only once. If the first instruction is a label in the form "TaskX:", then the INIT part is absent and the execution starts with Task1. Please note that a DEFINE with a negation creates an INIT part.

```
wakeup 2      ; this instruction is in the INIT part

Task1:
  y1=x1
  wait 1000
  ; hidden GOTO which restarts Task1
  ; this task keeps copying X1 to Y1, with pauses

Task2:
  y2=seconds and l<>0      ; "seconds" is a resource counting seconds
  ; another hidden GOTO to restart Task2
  ; this task blinks y2 every two seconds
```

The program above shows two tasks. Please note the "wakeup 2" instruction, which activates Task2: at start, all the tasks except Task1 are suspended.

Program format

An EZ-Red program is made by lines of text; empty lines are ignored (can be used to improve readability). The compiler is not case sensitive, any identifier can be written if free form.

Comments

```
; this is a comment. The next line will be ignored because it is empty.

y1=x1      ; this is another comment (the part after the semicolon)
[
this is a comment made by
several lines
]
```

The program can contains comments, beginning with a semicolon and extending to the end of line, or multi-line comments started with a opening square bracket and ended with a closing square bracket:

Identifiers

Identifiers are words invented by the programmer (or the compiler) to give a name to resources like inputs, outputs, variables, labels and so on. They must start with a letter or an underscore, followed by any mix of letters, underscores and digits.

It is not possible to declare two times the same identifier; several identifiers (all the resources, plus "ON", "OFF", "TRUE" and "FALSE") are already existent because the compiler declares them.

Numbers (numeric literals)

The language only has unsigned integers, so literal constants must be formed only by digits, without dots or other signs. However, it is possible to specify constants in binary or hexadecimal notation, using the suffixes "0b" (zero-B) for binary and "0x" (zero-X) for hexadecimal:


```
ylampmask1 = 0b1011      ; binary number 1011 (11 in decimal)
bcw_chr = 0x55           ; hex 55 (01010101 in binary)
```

Labels (declaration)

A line can start with a label, which is an identifier followed by a colon. If present, a label must be first in a line. As identifiers can not be redefined, there can not be two identical labels. The line containing a label can prosecute with an instruction or a comment, or both, but generally they contain only the label for readability.

Labels in the form "TaskX", where X is a number, are special because declare a task. The first declared task must be "Task1" and every new task must be numerically consecutive to the previous. The maximum number of tasks is 16.

Inside IF/THEN/ELSE blocks it is not possible to declare labels, but is possible to reference (target) them, using GOTOs.

DEFINE (declaration)

The DEFINE declaration allow to give a name to a resource or a (numeric) constant:

```
DEFINE motor      y2      ; assign a name (and this description) to Y2
DEFINE limit_switch !x1   ; inputs and outputs can be declared "negated"
DEFINE time_to_wait 2000  ; this is a constant declaration
```

If the negation notation "!" is used in a DEFINE, the compiler puts an invisible instruction in the INIT part; this instruction reverses the logic of the defined identifier, AND ALSO the original resource.

A comment on the same line after the DEFINE associates a description to the identifier created. The same is true if the comment is alone in the previous line (directly above).

DECLARE (declaration)

DECLARE creates a variable, of type BIT or INTEGER, and assigns a value to it:

```
; DECLARE [LOG] [R|DT] id = expression
declare time = 20           ; creates an integer, with value 20
declare log x1maybex2 = x1 or x2
    ; creates a bit e assigns it (only when executed) the OR of X1 and X2
    ; every change in x1maybex2 will be notified to the PC

define start x1

; this comment will be associated to "running" declared below
declare r running = running or start
declare dt cycles_count = cycles_count+1
    ; if the expression contains the identifier to be declared,
    ; it is mandatory to qualify with modifiers "R" or "DT"
```

The LOG modifier is only valid in BIT declarations. Like DEFINE, a comment immediately before or after in the same line can be used to associate a description to the symbolic name.

Instructions (statements)

Instructions differ from declarations because they generate code (but negated DEFINEs, DECLAREs and task labels are exceptions). There are three kinds of instructions: *assignments*, *commands* and *flow control* instructions. Most instructions need *expressions*, which are a combination of identifiers, constants and operators.

Assignments

An assignment has form "resource = expression". Practically every operation targeting the hardware is made by an assignment: to turn on the first power output "Y1 = ON" is used and to bring 10 volts to analog output 2 "AOUT2 = 255" is used.

The part to the right of the equal sign must be an expression that can be a simple constant, the name of a resource or a combination of many terms. The expressions are explained in detail below. The data type yielded by the expression (bit or integer) must coincide to the one of the resource at the left of the equal sign.

When assigning a power output Y1..Y8, the real state of the output is only updated when Task1 jumps backward or executes a WAIT. However, the internal state always reflect the last write. To update immediately the hardware of an output, use the following syntax:

```
Y2 <= ON           ; immediate update
```

Please refer to "I/O: synchronous or asynchronous" for more information.

Expressions

Expressions are a series of values combined by operators. Values can be literal constants or resource identifiers. An expression does the calculation defined by its operators and returns a single result. Example of expression are "1+5", "cycles+1", "x1 OR x2", "x1 or ain1>ain2+4".

Expressions must start with a "value", like a number or an identifier, and prosecute with couples of "operator value". A "value" can be an expression itself, surrounded by parentheses. For example:

```
3                ; the simplest one, a single literal
x1               ; a single resource
x1 or x2         ; two terms combined with OR
temp+sensor/2    ; three terms (and hence two operators)
(temp+sensor)/2  ; two terms, the first is in turn an expression
```

Every operator accepts only a certain type of operands (terms); for example, the "+" operator only accepts integers, however there are some which exist in a double version - logic, operating on bits, and *bitwise*, operating on integers. Most operators accept a certain type and return the same, but there are others which accept integers and return boolean (bit), like "<" which compares two integers and tells whether the first is less than the second. Different operators have different precedences: "12+4 / 2" is 14 (not 8!), because the division operator "/" has higher precedence than sum "+".

In order of growing precedence, these are the available operators:

Logic OR	Logic XOR	= (bit)	<> (bit)			boolean/bit data logic operations
Logic AND						
<	>	<=	>=	= (int)	<> (int)	int data, bit result
+ (plus)	- (minus)					arithmetic operators
* (mult.)	/ (divis.)					
Bitwise OR	Bitwise XOR					bitwise operators
Bitwise AND						

EZ-Red supports an alternative syntax: instead of “x1 or x2 or x3” it is possible to write “or x1 x2 x3”

Operators

For boolean (bit) data type there are the following operators:

Operator	Operands	Result	Description
AND	bit, bit	bit	Logic AND. Result is TRUE if both operands are TRUE.
OR	bit, bit	bit	Logic OR. Results is TRUE if either or both operands are TRUE.
XOR, <>	bit, bit	bit	Logic XOR. Result is TRUE if the operands are different.
=	bit, bit	bit	Equality. Result is TRUE if the operands are equal.

For numerical (integer) data the following operators are available:

Operator	Operands	Result	Description
AND	num, num	integer	Bitwise AND. Every bit of the first operand is ANDed with the corresponding bit of the second. The result contains those 16 bits.
OR	num, num	integer	Bitwise OR.
XOR	num, num	integer	Bitwise XOR.
=	num, num	bit	Equality. Result is TRUE if the numbers are equal.
<>	num, num	bit	Diversity. Result is TRUE if numbers are different.
<	num, num	bit	Compare. Results TRUE if the first operand is less than the second.
<=	num, num	bit	Compare. TRUE if the first number is less or equal than the second.
>	num, num	bit	Compare. TRUE if the first operand is greater than the second.
>=	num, num	bit	Compare. TRUE if the first number is equal or greater than the second.
+, -	num, num	integer	Arithmetic addition / subtraction
*, /	num, num	integer	Arithmetic multiplication / integer division

NOT (operator / function)

The operators listed above accept two operands. The NOT operator (or function) negates the expression to its right. If the expression is of BIT type, a negated bit is returned; if the expression is of type integer a complement to two negation is performed.

Special operators (modifiers) for bits: /, \, ^, !

In the application field of EZ-Red often is needed to negate a signal or manage its edges. To facilitate these, there are a few modifiers to be put immediately before an identifier:

Modifier	Description
/	Rising edge (true if bit is ON but was OFF before)
\	Falling edge
^	Any edge (true if the bit is different than before)

The expression “/X1” is true if, when evaluated, X1 is ON but in a previous evaluation it was OFF.

Consider this example: Y1 must be turned ON when a button connected to X1 is depressed (Y1 will be turned off elsewhere); if the button stays pressed Y1 must not be touched again:

```
if /X1 then Y1=on
```

On every execution of “/X1” a comparison is made between X1 and the previous reading of X1. The new reading is stored internally, for the next execution. In the following fragment:

```
if /X1 then Y1=on
if /X1 then timersec2 = 15
```

there are two "previous states", so there are two internal memories - one for every expression, even if they both refer to the same resource X1. This is different from what PLCs do normally.

It must also be noted that:

```
Y1=/X1
```

turns on Y1 for probably very short time (the time between an evaluation and the next).

Typical usage for this edge notation is in IF-THEN construct like:

```
if /button then timersec1=10
```

or in self-sustaining expressions or inversion switches like:

```
running = running xor /start
```

which inverts the state of "running" every time the switch "start" is pressed.

WAIT instruction

WAIT is a complex instruction used to wait for 0 to 4 events, within a maximum allowed time. The form with zero specified events is:

```
WAIT [timeout_in_milliseconds]
```

and results in a temporary suspension of the task for the specified time. The timeout can be specified with a constant or a resource name, but not an expression (which can be calculated beforehand). If no timeout is specified, or if its value is zero, the effect is the same as the SUSPEND instruction.

With or without a timeout, it is possible to indicate up to four events to wait for. An event is a particular state of a bit (on or off): so a bit must be indicated, optionally negated with "!". With no negation, the bit must be ON to satisfy the event; if negated, it must be OFF. When any of the specified bits reaches the indicated state, the WAIT is satisfied; these events are combined by a logic OR:

```

WAIT                ; suspend the current task
WAIT 0              ; suspend the current task
WAIT 1000           ; waits for a second

WAIT X1 X2          ; wait for X1 or X2 to become ON
WAIT !TMS1          ; wait for timer T1 to expire

WAIT 100 X1         ; wait for X1 on, for 100 ms

WAIT 1000 X1 X2 !TMS1 !Y2
    ; wait, for a maximum of a second, that X1 or X2
    ; raise to ON or TMS1 or Y2 fall to off.

WAIT start          ; wait forever for start to be ON
    
```

When a WAIT instruction has some event to wait, but no timeout or with value 0, the wait time is infinite.

A note on waiting times: if a timeout of *n* is specified, the actual time goes from *n-1* to *n* milliseconds. For example, "WAIT 1" waits *at maximum* 1 millisecond. To wait *at least* 1 millisecond, "WAIT 2" must be written. The motivation for this is as follows. Suppose to generate a square wave with 2 ms of period; the following code would not work:

```

Task4:
    ; generate a square wave with period of 2 ms
    y2 <= on
    wait 1
    y2 <= off
    wait 1
    
```

Instructions like "y2<=on", "y2<=off" take time to be executed, and this time must be added to the 2 milliseconds specified by the WAIT instructions. The generated square wave would not have a 2 ms period, but slightly more. EZ-Red instead generates exactly 500 hertz (period=2 ms); this happens because the WAIT instruction is synchronized to the internal 1 KHz tick: the effect is to delay at most 1 ms less than what specified, but the time saved is used to execute the other instructions.

WAITREMAIN (residual time after a WAIT)

When WAIT terminates, the residual time is stored into WAITREMAIN_x, where x is the task number which executed the wait. This value is zero if a WAIT terminated because of timeout (no events happened), and it is different than zero if an event made the WAIT to exit. This resource can be used to measure time, or to execute more than a WAIT in cascade with a total time:

```

[ Task1 ]
    y1 = on
    wait 1000 ...
    wait waitremain1 ...
    y1=off
    
```

In the fragment above, Y1 stays on for no more than 1second, independently of the events specified in the WAIT instructions.

Main resources (inputs, outputs, general hardware)

"Resource" is a general term to indicate hardware parts, like inputs and outputs, but it comprises also virtual (software) devices and flags, options, variables and so on. Even the constants ON and OFF are, actually, implemented as resources. A descriptive name is assigned to resources: when they are single, names like FEEDBACKS or WDTOUTS is used; when there are several similar resources, like the digital inputs, a name is used with a trailing number, as with X1, X2, X3.

The following table shows a list of main resources. The Quantity column shows the number of available resources with that name; for example, there are eight digital inputs named from X1 to X8: the name is "X" and the quantity is 8. Later in this document more details and more resources will be described.

Name	Quantity	Description
TRUE, ON	-	Constant indicating logic high level: Y1=TRUE; Y2=ON.
FALSE, OFF	-	Constant indicating logic low level: Y1=False; Y2=Off
X	8	Digital inputs, from X1 to X8. Input status is filtered and can be inverted - see the specific section.
XINVERT	8	Sets the logic inversion of the corresponding X input.
XCOUNT	2	Counts rising edges of X1 and X2 inputs, from 0 to 65535.
FX	2	On/Off status of the two fast inputs. These inputs are designed for an encoder, but can be used like any other digital input. They don't implement filters and logic inversion, though.
FXCOUNTL FXCOUNTH	2	They count the rising edges of FX inputs. The value of FXCOUNTL goes from 0 to 65535 (16 bits), then it resets and 1 is added to FXCOUNTH.
Y	8	The eight power outputs, from Y1 to Y8. If set to ON (or TRUE) the output will have voltage (however there is an option to invert the logic).
YINVERT	8	Logic inversion for the corresponding Y power output.
YLAMPMASK	8	Resources to make an output blink with a pattern.
TIMERMS	6	16 bits timers counting milliseconds. See the specific paragraph.
TIMERSEC	6	16 bits timers counting seconds.
TIMERMIN	4	16 bits timers counting minutes.
TMS	6	On/Off state of the corresponding TIMERMSn.
TSEC	6	On/Off state of the corresponding TIMERSECn.
TMIN	4	On/Off state of the corresponding TIMERMINn.
MILLISECS	-	Free 16 bits counter in milliseconds. When 65535 is reached, it resets to 0.
CENTISECS	-	16 bits counter counting hundredths of seconds.
SECONDS	-	16 bits counter, counting seconds.
R	64	Internal relays (bit type), to be used as general boolean variables.
DT	64	Internal 16 bit memories - integer variables. To hold numbers from 0 to 65535.
AOUT	2	Analog outputs 0-10V. Their value goes from 0 (0 volts) to 255 (10 volts).
AIN	2	Analog inputs. They read the voltage at the input, and return a number from 0 (0 volts) and 255 (10 volts).
ENCODERL, ENCODERH	-	Position (value) of the optional encoder connected to FX inputs. Refer to the specific paragraph.

[Digital inputs X1..X8](#)

X1..X8 report the status of digital inputs 1..8, and have some options associated. The first one is the *logic*

inversion of the input. When using a limit switch, it is normal to use a normally closed contact, which brings 24 volts to an input. When the limit switch is engaged, the contact opens and the input goes low. In the program one could "DEFINE LIMITSWITCH X1", but in that case LIMITSWITCH would go low when the switch is engaged, so all the program would use an inverted logic. To have a lamp light on when the axis is at its limit, one should write "LAMP = LIMITSWITCH<>ON". It would be better to write "LAMP=LIMITSWITCH", and this becomes possible if the logic of X1 is inverted. It can be made with a DEFINE:

```
DEFINE limitswitch !x1
```

XINVERT (logic inversion of inputs)

The effect of the previous DEFINE is simply to set the XINVERT1 bit resource in the hidden INIT section of the program. In any moment, it is possible to modify XINVERT1..XINVERT8.

Please note that the "!" notation used in expression is a separate function. If the DEFINE above is in effect, writing "!limitswitch" in an expression would read X1 without any inversion.

XTHRESHOLD UP/DN (noise / anti bounce filter)

Reading a voltage on the inputs and transferring it to an X resource also depends on a software filter to reduce noises and bounces. By default there is no filter, and the inputs get sampled 1000 times a second; so a variation from 0 to 24 volts is noticed in a millisecond. Sometimes this is not desirable, perhaps because of noise. The XTHRESHOLDUP resource group defines, in milliseconds, the time a voltage must stay present before turning a logic input from OFF to ON. If the threshold is set to 5, for example, noisy pulses of 3 or 4 milliseconds are suppressed. The threshold however introduces some delay, necessary to verify the voltage is stable.

The same way as XTHRESHOLDUP, XTHRESHOLDDN sets the time to wait in a transition from voltage present to no voltage present. This is particularly useful to debounce normally open contacts connected between +24V and the input. When the contacts close, voltage is immediately brought to the input but, if the contacts are not clean, voltage can drop suddenly and the PLC could interpret this as a release of the contacts. In this case setting XTHRESHOLDDN would add stability (and delay) in the transition from ON to OFF.

XTHRESHOLDUP e XTHRESHOLDDN accept values from 0 to 127 (milliseconds).

XCOUNT1 and XCOUNT2 (rising edge counters)

Attached to the first two digital inputs, X1 and X2, there are hardware counter which detect and count rising edges of the incoming signals. Their names are XCOUNT1 e XCOUNT2, containing 16 bits values (from 0 to 65535). Please note that there is not filtering and no *logic inversion*.

Power outputs Y1..Y8

The power outputs Y1..Y8 are active when set to ON (or True), to have 24 volts, and turned off (no voltage) when written OFF or FALSE. Depending on the setting of AUTOUPDATEXY, a write instruction can have effect immediately or deferred (see I/O synchronous or asynchronous); in any case Y1..Y8 can be "read" as terms in expressions and always reflect the last assignment made, even if the value has not yet been transferred to the physical output.

YINVERT

Like the digital inputs, the digital outputs also can be logically inverted by setting the relevant YINVERTn to ON. A declaration like "DEFINE alarm !Y1" actually sets YINVERT1 to TRUE.

Feedback (outputs failure alarm) FB12, FB34, FB56, FB78, FBYTE

EZ-Red can detect, on power outputs, overload and open circuit (which can meaning a broken device).

When an Y output is low (no voltage), a broken circuit can be detected and the corresponding bit FBnn is set to TRUE. When an Y output is ON, an overload can be detected: current is limited and, after little time, the relevant

FB bit is set. Feedback circuitry manages outputs in couples; for Y1+Y2 there is a single FB12 bit; then FB34 for Y3 and Y4, and so on. The FBBYTE resource, in the first 4 low bits, comprehends all the FBnn feedbacks; with a single reading it is possible to detect electrical faults.

These feedbacks can trigger the watch-dog if they are enabled through WDTFBENAnn (12, 34, 56, 78) or the WDTFBBYTE resource. See the Watch-dog chapter for more details.

TIMERS: TIMERMS, TIMERSEC, TIMERMIN

EZ-Red has 16 16-bit timers, made of a counter (down-count) and a virtual contact (a bit). The counter can be read and written by program; it contains values from 0 TO 65535, and decrements to zero automatically. The corresponding bit is ON when the timer (counter) is still counting (its value is different than 0) and OFF if the counter is 0. When the program writes into the counter, the counting starts from the new value written (the timers are retriggerable), and the output bit is also updated.

There are 6 millisecond counters, from TIMERMS1 to TIMERMS6, and the corresponding output contacts are TMS1..TMS6. Writing 1000 into TIMERMS1, its output TMS1 goes immediately ON; for the time of a second (1000 milliseconds), the value of TIMERMS1 decreases down to zero, and finally the output contact TMS1 turns OFF. The maximum counting time of these timers is about 65 seconds.

There are then 6 timers which count seconds, from TIMERSEC1 to TIMERSEC6, and the corresponding outputs are TSEC1..TSEC6; their maximum counting time is more than 18 hours. Lastly 4 timers are available which count minutes: from TIMERMIN1 to TIMERMIN4, having as output respectively TMIN1..TMIN4; the maximum counting time is over 45 days.

Timers are easy to understand and deploy, especially combined with IF instructions. For example, the following fragment:

```
define      button      x1
define      lamp        y1

Task1:
  if /button timerms1 = 1000
    lamp = tms1
```

makes the lamp turn on for a second every time the button is pressed. If the button is pressed and then kept down, a single flash is produced. If the modifier "/" was not used, like:

```
if button timerms1 = 1000
```

the lamp turns on when the button is pressed, and turns off a second after the button is released.

FX inputs, FXCOUNT and encoder interface

The fast inputs FX1 and FX2, in addition to being normal inputs (without filters and logic inversion), have hardware counters which count rising edges.

FXCOUNTLx and FXCOUNTHx

For every rising edge of the FX input, the hardware counter increments; when the value reaches the hexadecimal number FFFFFFFF (over 4 billions), the counter resets to 0 (overflow). The counter can be set to any value with a normal assignment.

The counters are 32 bits wide, but the the PLC program only uses 16 bits. Every counter is hence broken in its high part (FXCOUNTH) and low part (FXCOUNTL). To counting up to 65535 the high part can be simply ignored.

Encoder

The fast inputs FX1 and FX2 can be connected to the A and B channels of a quadrature encoder. In this case EZ-Red traces the encoder position in a 32-bit register accessible by program. When the encoder makes a step in "+" direction, the register increments by one; if the register was already containing the hexadecimal number FFFFFFFF, it resets to 0 (overflow). Likewise, when the encoder makes a step in "-" direction the register decrements; if it is already 0, it becomes FFFFFFFF (underflow).

The register is 32 bits wide, but the PLC program only uses numbers of 16 bits, so the register is split in low (ENCODERL, the "L" stays for "low") and high part (ENCODERH, the "H" stays for "high"). If the positions are limited to the range 0 to 65535 the high part can be completely ignored.

Encoder preset

It is possible to do a zero or set of the encoder register (preset). Because the hardware register is 32 bits wide, but it is accessed in two 16-bits operations, it is mandatory to write the low part first, and then the high part:

```
; encoder preset
encoderL = 200
encoderH = 0      ; optional
```

As soon as the low part is written, the high part is automatically set to 1 (by hardware). This is not a problem if ENCODERH is not used in the cycle; if instead it is, the next instruction sets it to the desired value. This behavior is necessary because the encoder hardware works continuously, and it could detect a movement between the first and the second preset instruction, leading to erroneous results.

The PLC program does not use negative numbers, so instead of zeroing the encoder (for example in correspondence of a limit switch) it is better to preset it to a positive number, like 200 or 1000. This allow the encoder to decrement a few steps without causing an underflow.

Flow control (IF-THEN-ELSE and GOTO)

"Flow control" means the possibility to alter the rigid way instructions are executed, one after another from top to bottom of the text. Without flow control every task of EZ-Red would be like a Ladder.

The IF instructions permit to execute instructions only if some condition is verified; this is near to the way human beings think. In the simplest form, a single instruction is executed or not depending on a condition:

```
if x1=on then r2 = on
```

After the word "IF" a boolean expression (bit type) must be written, for example "ain1>128" or "timerms1<>0". After the word THEN a single command/instruction must be present.

It is possible to submit more than one instruction (a block) to a single boolean expression:

```
if x1 then
  r2 = on
  dt4 = 128
end
```

The compiler recognizes that there is a block instead of a single instruction because there the word THEN is last on its line. All the next lines, up to the one with END, are executed if the boolean expression (simply "x1" in this case) is TRUE; otherwise, no instruction is executed.

Sometimes this kind of IF is not sufficient; one could want to execute something from something else, depending on a condition:

```
if ain1>128 then aout1=64 else aout1=192
```

In the example above, either "aout1=64" or "aout1=192" are executed, depending on the value of ain1. The word ELSE introduces the instruction to execute when the IF expression yields FALSE. The ELSE clause is clearly optional, but even the THEN clause is:

```
if x1 then else dt1=0
```

is legal, but similar constructs are seldom used because inverting the test (condition) is prettier:

```
if not x1 then dt1=0
```

As for the THEN clause, even the ELSE part can be a block instead of a single instruction:

```
if r1 or x2 then y1=on else
  y1=off
  y2=on
end
```

It is easy to imagine that both THEN and ELSE clauses can be formed by a block. The following fragment:

```
if temperature<setpoint then
  res_1000watt = on
  res_2000watt = off
  if temperature+10 < setpoint then res_2000watt = on
end else
  res_2000watt = off
  if temperature > setpoint+10 then res_1000watt = off else res_1000watt = on
end
```

is a simple double-power oven control which tries to maintain temperature inside a certain range.

A block can contain any number of instructions, including other IFs, but not DEFINES or labels (which are not instructions but declarations). There is no limit on the number of nested IFs, but the program can get difficult to read if there are many nested blocks.

GOTO (unconditional jump)

IF-THEN-ELSE instructions allow to vary in some way the execution, but the instructions are still executed "top to bottom". Another way to alter this "top to bottom" execution is GOTO; simple and deprecated but sometimes necessary, it transfers execution to another part of the code, that must be correctly labeled:

```
IF alarm THEN goto exit
IF button=off THEN goto exit

; a complex part of the program...
...
...

exit:
...
```

The target label in the example is "exit:", written in the same way task labels are. The same code fragment could be written without using a GOTO, but perhaps it becomes less readable:

```
IF alarm=false THEN
  IF button=on THEN
    ; complex part of the program
    ...
    ...
  END
END

exit:
...
```

Jumps made by GOTOs should always jump in the same task, not another task or in the INIT section of the program.

labels must be unique in the program, exactly like any other identifier, but a single label can be targeted by more than one GOTO.

Special instructions and resources

Task control

An EZ-Red program can contain up to 16 Tasks. A task can be used, in addition to control a part of the application, to simulate new devices like counters or timers, and can even be used as a subroutine. There must be a way then to stop a task or restart it.

WAKEUP (task activation)

When a program starts, all the tasks except the first (Task1) are suspended (sleeping). This way it is possible to set up the environment (Init) without interferences from concurrent tasks. To activate a task the WAKEUP instruction is used, followed by the number of the task. The number must be in the range 1 to 16, and expressed by a constant or an identifier created by a DEFINE referring to a literal:

```
WAKEUP 2      ; activates task 2
; or...

DEFINE  pid  4
...
wakeup pid
```

If the target task is already active, the instruction has no effect; otherwise, the task continues execution from the point where it was stopped. While a program is starting up, all the tasks are sleeping on their first instruction.

RESTART (start again from the beginning)

The RESTART instruction is similar to WAKEUP, but it resets the task to its first instruction:

```
RESTART 2      ; resets task 2 (suspended or not)
                ; in any case, Task2 restarts from begin

Task3:
...
RESTART
...
```

The execution point of the task is moved to the beginning and, if the indicated task is stopped (suspended), it is re-activated.

The argument of RESTART is optional: if omitted, the current task is implicit. Of course, a suspended task can not restart by itself.

SUSPEND

A task can be stopped (suspended) by the SUSPEND instruction. The indicated task stops in the point it is executing, and a later WAKEUP will make the task proceed from there. The argument to SUSPEND is optional and, if omitted, the current task is implicit. This is the best way to suspend a task, because the point of stopping is known. In other words, it is better for a task to let it suspend itself instead of stopping it in an unknown position:

```

; task 2 generates a delayed pulse
Task2:
  wait 1000          ; wait for a second
  Y1 = ON
  wait 500
  Y1 = OFF
  suspend

Task3:
  ; to generate a delayed pulse, simply do:
  wakeup 2
    
```

The above fragment shows how simple is to generate a delayed pulse without using a timer.

CYCLERUN

The CYCLERUN bit is ON when the PLC executes a cycle: its main purpose is to let the PC know if a cycle is running. However, the PLC program can write to this bit and, if it writes OFF, the cycle stops.

Watch-dog

A Watch-dog is a mechanism that monitors some aspects of a system in order to determine whether some failure is present; if some failure is detected the system is set to a known and secure state (a state where no damage can occur).

In the case of industrial automation, for example in controlling the temperature of an oven, the insecure state is when the oven is turned on: if the system stops to control, the temperature can raise too much.

With EZ-Red, the watch-dog can be used to monitor the following aspects of the system:

- 1) communication with the PC (if stops, then a part of the system is compromised)
- 2) the correct work of the power outputs (overload and broken circuit)
- 3) internal power supplies of EZ-Red

If configured to monitor one or more of the described aspects, and a failure is detected, the watch-dog fires and makes the following:

- 1) sets the power outputs to a precise condition (configurable), considered "safe"
- 2) sets the analog outputs to a certain level (configurable)
- 3) optionally stops the cycle execution

These three things above are performed 100 times a second: if the PC modifies the outputs while the watch-dog is active (fired), no more than a hundredth of a second later the watch-dog sets them again.

If the cycle stop option (WDTSTOPSCYCLE) is off, and a cycle is running, then the only effect of the watch-dog is to set the WDTFIRED bit; the outputs are leaved untouched because in that case it is responsibility of the program or the computer to take appropriate actions to correct the problem.

WDTFIRED e WDTSTOPSCYCLE

WDTFIRED is a bit indicating that the watch-dog is fired; it is read-write: setting it to TRUE makes the watch-dog fire, and setting it to FALSE cancels the fired condition. Canceling the watch-dog is not useful if the conditions which fired it persist – before canceling the watch-dog the failures must be corrected or the part of the watch-dog which detects those failures must be disabled.

WDTSTOPSCYCLE is a bit which determines whether, when the watch-dog fires, the running cycle has to be

stopped or not. If WDTSTOPSCYCLE is FALSE the cycle is not stopped: rather, it is the cycle which have responsibility to manage the failure. At EZ-Red start-up this bit is TRUE.

WDTOUTS, WDTAOUT1-2 (watch-dog outputs pattern)

These three resources contain the values to be imposed on the outputs when the watch-dog fires. WDTOUTS is a byte, like YBYTE; it contains a bit for every Y output, and it is continuously copied to YBYTE when the watch-dog is in fired state. The same happens for WDTAOUT1, copied to AOUT1, and WDTAOUT2, destined to AOUT2.

The default values (at start-up) are 192 for WDTOUTS (Y1 to Y5 off, Y6 and Y7 on), and 0 for WDTAOUT1 and WDTAOUT2.

WDTTIME (PC communication time-out)

This resource is of integer type and contains a number, from 0 to 65535, indicating the time, in milliseconds, EZ-Red waits for a communication from PC. If this time expires, the watch-dog fires. Setting WDTTIME to 1000, for example, makes EZ-Red expect a packet from PC at least every second; if a packet delays more than a second from the previous one, the watch-dog fires. This time-out should be planned basing on the application requirements, but the responsiveness of a PC, especially if running some windowed GUI, is not very high and 1000 milliseconds could be too low.

When WDTTIME is set to 0 (default at start-up) no time-out is in effect: the watch-dog will not fire because a failure in the communication with the PC.

WDTFBENA12-34-56-78 e WDTFBBYTE (power outputs monitoring)

These boolean resources (or they group in the first 4 bit of WDTFBBYTE) connect the power outputs feedbacks to the watch-dog: if a failure feedback (FB12, FB34...) goes TRUE, and the corresponding enable bit (WDTFBENA12, WDTFBENA34...) is also true, the watch-dog fires.

Feedbacks detect an excessive current when the output is ON, and an open circuit when the output is OFF. The power outputs which are not connected should be kept in ON state to prevent their feedback bit to become TRUE. Alternatively, it is possible to let the feedbacks bit go TRUE, and avoid to enable the corresponding watch-dog enable bit.

Feedback bit (FB12, FB34...) and the related enable bits refer to couple of outputs 1-2, 3-4, 5-6, 7-8.

Interaction between watch-dog, PC and PLC cycle

When the watch-dog is actively firing, the “safe” configurations for the outputs is set 100 times a second, even if the PC continuously tries to modify them. The PC should correct the failures which fire the watch-dog or disable their detection, through WDTTIME / WDTFBBYTE, before resetting the watch-dog by turning off the WDTFIRED bit.

If a PLC cycle is running, and WDTSTOPSCYCLE is FALSE, a fired watch-dog does not stop the cycle and the latter can (or should) take care of the situation: one of the possible ways is to reserve a task just for that, like the following example:

```

Task3:
; manage the watch-dog intervention
WAIT WDTFIRED      ; wait until the watch-dog fires

; if execution reach here, it is because the watch-dog has fired
SUSPEND 1          ; stop the task which modifies outputs
SUSPEND 2
... ; other actions
YBYTE = 128        ; set the output to a "safe" pattern
... ; other actions
    
```

Task3 waits for the watch-dog to fire (of course this can also never happen). If the watch-dog fires, with a minimal latency Task3 resumes execution and, in this example, first suspends the main tasks (to avoid interferences) and then ts the outputs in a convenient, “safe” configuration; in this example an emergency lamp is connected to the power output 8. Of course Task3 must be activated in the Init section or by Task1 with "WAKEUP 3".

LOG instruction (data acquisition and debug)

When a PC is connected to a EZ-Red, it can interact in many ways on inputs, outputs and internal data. However the PC often has long latency times which, added to the latency of the USB interface, can be excessive; this is why EZ-Red contains a programmable PLC – to get fast and predictable reaction times. When it is desired to acquire some data on a certain event, EZ-Red can do it with reasonable speed, but not always the PC can be fast enough, because of the latencies said before. In these cases the LOG instruction is useful: EZ-Red makes the acquisition of data, and sends it to the PC with a circular buffer (a transit memory) which eliminates the latency problems. The LOG instruction sends data to the PC, a 16-bits integer or a single bit, through an arbitrary “channel” chosen between 1 to 255. The complete syntax is:

```
LOG channel int_or_bit_value
```

The PC, connected via USB and using the DLL library with the Windows messaging system, or alternatively using the terminal emulation mode, receives an asynchronous communication which does not interfere with the ongoing dialog. This function can also help debugging cycles.

REPORTBACK, SENDTOPC (inputs status transmission)

The REPORTBACK bit, if set to TRUE, automatically transmits a packet to the PC on every input status change; the packet also contains the encoder position. The same communication can be initiated by the cycle by writing TRUE in the SENDTOPC bit: this bit is automatically reset when the transmission has taken place.

Special functions for configuration, protection and diagnostics

EZ-Red has some features that can be used in the cycle to for protection, diagnostics and configuration.

CONFIGxxx (BIT, CHR, WRD)

There are 16 words 16 bits wide, named CONFIGWRD1..16, which are non-volatile: their content is preserved across power-off/power-on cycles because it is written in the internal flash memory of the module.

For convenience, these 16 words are also accessible in bytes (8-bits wide) and some of them even in single bits. The byte resources are CONFIGCHR1..32, overlay on CONFIGWRDn: when writing to either CONFIGWRDn or CONFIGCHRn the other reflects immediately the new data. The disposition is little-endian - CONFIGWRD1 contains CONFIGCHR1 in its low part and CONFIGCHR2 in the high part; WRD2 contains CHR3 and CHR4, and so on. Setting CONFIGWRD1=1, implicitly CONFIGCHR1 becomes 1 and CONFIGCHR2 becomes 0. The

first 4 resources, CONFIGCHR1..4, corresponding to CONFIGWRD1..2, are also overlay with single bits named CONFIGBIT1..32. The following table shows the memory map:

INT (16 bit)	BYTE (8 bit)	BIT
CONFIGWRD1	CONFIGCHR1	CONFIGBIT1 ... up to ... CONFIGBIT8
	CONFIGCHR2	CONFIGBIT9 ... up to ... CONFIGBIT16
CONFIGWRD2	CONFIGCHR3	CONFIGBIT17 ... up to ... CONFIGBIT24
	CONFIGCHR4	CONFIGBIT25 ... up to ... CONFIGBIT32
CONFIGWRD3	CONFIGCHR5-6	NOT BIT-ADDRESSABLE
CONFIGWRD4	CONFIGCHR7-8	
CONFIGWRD5	CONFIGCHR5-6	
CONFIGWRD6	CONFIGCHR5-6	
CONFIGWRD7	CONFIGCHR5-6	
CONFIGWRD8	CONFIGCHR5-6	
CONFIGWRD9	CONFIGCHR5-6	
CONFIGWRD10	CONFIGCHR5-6	
CONFIGWRD11	CONFIGCHR5-6	
CONFIGWRD12	CONFIGCHR5-6	
CONFIGWRD13	CONFIGCHR5-6	
CONFIGWRD14	CONFIGCHR5-6	
CONFIGWRD15	CONFIGCHR5-6	
CONFIGWRD16	CONFIGCHR5-6	

BCW_xxx (BIT, CHR, WRD)

These special resources manipulate bits, bytes (char) and words (16 bits) without using logic instructions. Like CONFIGXXX, several resources share the same memory space, so it is possible to convert integers to bytes and bits. There are 2 16-bits integer, BCW_WRD1 e BCW_WRD2, visible as BCW_CHRn (1 to 4) and BCW_BITn from 1 to 32.

DISABLEUSB

If this bit is set (ON or TRUE), EZ-Red refuses to communicate and only replies to these requests:

- identification (model and firmware version)
- parameters (configuration) read and write
- unlock (with password)

FIRSTRUN

This bit indicates whether Task1 is executing its cycle for the first time (when TRUE), or it has at least once executed a back jump. This bit is inherited from PLCs, but is not really useful because in EZ-Red is more easy and correct to use the INIT part if the cycle.

FORCEDXS

The PC, via a special function, can virtually “force” a digital input in a certain state; when an input is forced EZ-Red behaves like it is really reading the forced state instead of the true one. The FORCEDXS bit indicates that one or more inputs are in forced state.

PCCONNECTED

This bit indicates to the PLC program that a PC is connected to the USB port and it is actively communicating. This bit automatically resets to FALSE after 60 seconds from the last received packet from USB.

PWDPROTECT

This bit is ON when a protection password has been set to write protect the PLC program. When PWDPROTECT is TRUE the program can not be modified or saved in the internal flash. The program can reset this bit, but not the password; at the next power-on if a password is stored the bit will be set ON again. It is also possible for the program to set this bit even if there is no stored password: the EZ-Red will behave as if a password is stored, until the next power-on.

Example programs

Some program can help to see the language features. The following examples are didactic, they don't aim to be perfect or even work correctly.

Press with safety pushbuttons (procedural)

This application consists of a press commanded by two switches; the cycle can start only if the two buttons are pressed after both have been released. The switches are normally open type and, when pressed, bring voltage to the connected EZ-Red input. To eliminate bounces XTRESHOLDxx are set: as the buttons are normally open, it is important to avoid that a bad contact is interpreted as a release, so the “DN” time (XTHRESHOLDDN) is the one to be set.

This example is written in procedural form. There are other ways to write it, using less IFs e GOTOs.

```

; Press control

define press    Y3
define btn1     x1
define btn2     x2

XTHRESHOLDUP1 = 40      ; values are ms
XTHRESHOLDUP2 = 40
XTHRESHOLDDN1 = 100
XTHRESHOLDDN2 = 100

Release:
  press = OFF
  if btn1 or btn2 then goto release

Standby:
  if btn1 and btn2 then goto crunchit
  goto standby

Crunchit:
  if not btn1 then goto release
  if not btn2 then goto release
  press = ON
  goto crunchit

```

Slide with button and two limit switches (functional)

In this application there is a slide, moved by a motor in two directions – forward or backward. Two limit switches bound the movement. The slide is normally retracted, on the back limit switch. A push button allow to push the slide forward, up to the forward limit switch; as soon as the command ceases the slide must come back. If the button (command) stays pressed for more than 10 seconds, the slide must come back. The program is written in two versions, *functional* and *procedural*.

The functional version is a sequence of assignments, in a way similar to ladder. The first part of the program

defines a few mnemonics for the utilized resources:

```

; Slide control

define max_time      10000 ; validity time if the command

define ok_backward  x1      ; NO back limit
define ok_forward   x2      ; NO forward limit
define button_forw  x3      ; slide forward button
define mot_forward  y1      ; motor forward command
define mot_backward y2      ; motor backward command

define timer timerms1
define com_valid    tms1    ; command validity timer
                        ; indicates the command is not too much long
    
```

The second part of the program executes the cycle:

```

; commands can last 10 seconds (10000 ms)
if /button_forw then timer = max_time
if !button_forw then timer = 0
    ; now com_valid tells if button press (command) is valid

; go forward if the command is valid, and...
; the limit switch is not engaged
mot_forward = com_valid and ok_forward

; go backward if there is no command...
; until the back limit switch
mot_backward = ok_backward and not com_valid
    ; or else: mot_backward = ok_backward and !com_valid
    
```

This program is functional because there are no real IF instructions: the only used one is for loading the timer, with a rising edge notation – it is a translation of a ladder branch.

This program, as is, actually does what is requested with “command must last 10 seconds maximum”. However, it is possible to release shortly the button to make the slide never return retracted. It could be desired that the slide is completely retracted before a new cycle can begin. To do this a single instruction has to be changed:

```
if !button_forw then timer = 0
```

can be modified in:

```
if mot_backward or !button_forw then timer = 0
```

Slide with button and two limit switches (procedural)

The following program is the procedural version, which analyzes the various phases of the cycle instead of expressing the outputs basing on the inputs without knowledge of the phase. Giving the same definitions as

before, except the timer which is no more needed:

```

; Slide control

define max_time      10000 ; validity time if the command

define ok_backward  x1      ; NO back limit
define ok_forward   x2      ; NO forward limit
define button_forw  x3      ; slide forward button
define mot_forward  y1      ; motor forward command
define mot_backward y2      ; motor backward command
    
```

the program describes one for one the various stages of the cycle:

```

; in stand-by, slide must go back until the limit switch
Standby:
    mot_forward = OFF
    mot_backward = ok_backward
    if button_forw and !mot_backward then goto forward
    goto standby

Forward:
    mot_backward = OFF
    mot_forward = ON

; wait for: timeout, limit, or command fall
wait max_time !ok_forward !button_forw
if waitremain1=0 then goto standby ; time out
if !button_forw then goto standby ; no command
; button pressed and still in time

; go to the limit, while there is time
wait waitremain1 !ok_forward !button_forw
if waitremain1=0 then goto standby ; time out
if !button_forw then goto standby ; no command

; got the limit switch
mot_forward = OFF
wait waitremain1 !!button_forw

goto standby
[ this GOTO is not required, the cycle restarts anyway ]
    
```

The program is longer, but analyzes deeply each single phase.

Slide, procedural, using two tasks

By taking advantage of the multi task capabilities of EZ-Red, it is possible to write this program in a different way. Knowing that releasing the button should retract the slide, a task can wait this event and restart Task1. The listing becomes more readable:

```
[ use the previous declarations ]

Task1:
  mot_forward = OFF
  mot_backward = ON
  wait !ok_backward
  mot_backward = OFF
  wait button_forw

  mot_forward = ON
  restart 2 ; task2 monitors the button release

; wait for timeout or limit switch
wait max_time !ok_forward
mot_forward = OFF
wait waitremain1
restart

Task2:
wait !button_forw
restart 1
suspend ; avoid to restart Task1 over and over
```

Motor with encoder control

In this application a motor is driven with forward and back commands, and speed regulated by a 0-10V signal. When the START button is pressed the motor makes a forward trip to a predetermined position or until the button is released. When the button is released, the motor goes back to the limit switch.

The first part of the code contains a few DEFINES:

```
; motor with encoder control

define position    28414      ; (example)

define mot_forward y1      ; "motor forward" command
define mot_backward y2     ; "motor backward" command
define start      x1       ; start button
define limit      !x2     ; limit switch (off=limit)
define speed      aout1    ; 0-10V command for speed control
```

The second part is the INIT: the motor goes out of the limit switch and then rewind slowly:

```

Init:
  ; exit from limit switch
  speed=10
  mot_forward = ON;
  wait 5000 !limit          ; exit from limit, or timeout
  if limit then goto error  ; failed to leave the limit switch

  ; go back to make a zero on the limit switch
  mot_forward = OFF
  mot_backward = TRUE
  wait 15000 limit
  mot_backward = OFF
  encoder1 = 200          ; preset to position 200
  if limit then goto cycle ; ok, ready to proceed

Error:
  wdtfired = ON;        ; this stops everything
    
```

The main part of the program only takes care to send motor forward and backward, looking at the START button. Speed regulation is demanded to Task2; if the button is released before reaching the final position, a smooth inversion has to be made with two acceleration ramps; the deceleration is made by the main task, to show an alternative way:

```

Task1:
Cycle:
  mot_backward = OFF
  wait start          ; wait for START button

  speed=10           ; go forward, with a ramp
  mot_forward = ON
  restart 2          ; task 2 controls ramp and stop
  wait not start
  suspend 2

  ; go back, with smooth inversion
decel:
  if speed > 10 then ; decrease down to 10
    speed = speed-1
    wait 10
    goto decel
  end

  mot_forward=off
  mot_backward=on

  wakeup 2
  wait limit
    
```

The task number 2 does start-stop ramps. Please note that while Task2 is running, Task1 is not stopped – it waits for the button to be released. Task number 2 is this:

```
Task2:
; go to the programmed position or 200 (zero)
; if near to target, decelerate – otherwise accelerate
if mot_forward then dt1 = position – encoderL else
  dt1 = encoderL
  if dt1 > 200 then dt1 = dt1-200
end
if dt1 < 900 then
  speed = dt1 / 5 + 5          ; near stop, decelerate
  if dt1<5 then
    if mot_forward then mot_forward=off
  end
end else
  if speed<200 then          ; if possible, increase speed
    speed=speed+5
    wait 5
  end
end
end
```



XON ELECTRONICS SRL
WWW.XONELECTRONICS.IT
INFO@XONELECTRONICS.IT

Internet product page is at <http://www.xonelectronics.it/prodotti/industriali/EZ-Red>

Please report any error or imprecision to web@xonelectronics.it