Get Going With WESCL

Language Documentation

Damian Wróbel

Get Going With WESCL: Language Documentation

by Damian Wróbel

Copyright © 2000,2001 by NETICA

Acknowledgement: Thanks to Christoph Buchala <netica@pro.onet.pl> for his reviews of this material and bug-findings escapades into WESCL.

Revision History

Revision 1.0	2000-09-18	Revised by: dwrobel@ertel.com.pl
Initial revision.		
Revision 1.2	2000-10-11	Revised by: dwrobel@ertel.com.pl
Public release. Including materials for the WESCL version 2.11.		
Revision 1.3	2001-01-08	Revised by: dwrobel@ertel.com.pl
Internal release. Including materials for the WESCL version 2.18.		

Table of Contents

1. Introduction 6
1.1. What is WESCL 6
1.2. How it works? 6
2. Basic Language Features 7
2.1. The Program 7
2.2. Comments 7
2.3. Values 7
2.3.1. Strings with backslash sequences 7
2.4. Variables 8
2.5. Data Type Conversions 9
2.6. Operators 9
2.6.1. Operator Precedence 11
2.7. Instructions 12
2.7.1. Expressions 12
2.7.2. Actions 13
2.7.2.1. String assignment actions 14
2.8. Working with instructions 14
2.9. Aliases 20
3. Special registers 21
3.1. Counter registers 21
3.2. Status registers 21
3.2.1. First loop indicator 21
3.2.2. Date and time registers 22
3.2.3. Hardware counter 22
3.2.4. Division by zero 23
4. Interaction between WESCL and HTML-browser 24
4.1. HTML-browser to WESCL communication 24
4.2. WESCL to HTML-browser communication 24
4.2.1. V register 25
4.2.2. E register 26

4.3. Compile WESCL programs 27
5. WESCL Communication Agent (WCA) 28
5.1. Circular Buffer 28
5.2. Message Structure 29
5.2.1. Message Address 29
5.2.2. Message Body 30
5.2.3. Message Attachment 30
5.3. Sending Messages 31
5.3.1. Return Codes 36
5.4. Assigned Registers 38
5.4.1. Argument Registers 38
5.4.2. Control Registers 39
5.4.3. Status Registers 39
5.4.4. Data Registers 40
A. WESCL grammar 41

List of Examples

2-1. Expression examples 13
2-2. Assignment action type 13
2-3. A variable name action type 13
2-4. Multi action examples 14
2-5. String actions examples 14
2-6. Instruction with a single assignment action....15
2-7. Instruction with multi action....15
2-8. Variable name action 16
2-9. Instruction with a multi variable name action 16
2-10. Instruction with a mixed mode action 17
2-11. Instruction with a complex mixed mode action....18
2-12. Instruction with a string assignment action 19
3-1. Using counters 21
3-2. Using first loop indicator 22
4-1. Setting register by HTML-browser 24
4-2. Reading WESCL variables 25
4-3. Using V register....25
4-4. Using E register 26
5-1. Sending simple message....33
5-2. Sending complex message....34

Chapter 1. Introduction

1.1. What is WESCL

WESCL is an abbreviation for *Web Embed Simple Control Language* and is designed for:

- Creating simple control processes over WebEmbed
- Supporting visualization processes of the controlled object.

1.2. How it works?

WESCL runs as a real-time task, which processes the instructions that have been loaded into the program memory area from the `"auto.scl"` file after its compilation. The program runs in a continuous loop, which consists of the following major phases:

- Begin-of-loop
- Read input signals
- Execute instructions
- Decrement counters
- Write output signals
- End-of-loop.

Chapter 2. Basic Language Features

2.1. The Program

WESCL program is placed in the plain-text `"auto.scl"` file located in the `/home` directory. Any comments or blank characters *(tabulator, new-line)* are ignored.

After successful compilation program starts working (executes sequence of instructions) at every 100ms.

2.2. Comments

WESCL supports the Java-style comments:

- Comments that span multiple lines are preceded by /* and followed by */
- Comments on a single line are preceded by a double-slash //

2.3. Values

WESCL recognizes the following types of values:

- Logical (Boolean) *true*, *false*
- Signed Integers in range: -32768..32767
- Strings such as "Hello!".

2.3.1. Strings with backslash sequences

Strings can use full-formated C-strings including backslash sequences as follows:

Special char	Description	Example
\n	Newline character	"First.\nSecond."
\r	Carriage return character	"1.\n\r2.\n\r"
\t	Tab character	1.\tStart\n2.\tEnd."
\\	Backslash	"/\\/\\"
\0nnn	Matches character that has octal value nnn	"NewLine\012\015"
\0xnn	Matches character that has hexadecimal value nn	"NewLine\0x0a\0x0d"

2.4. Variables

WESCL operates on registers. The following table shows all currently available registers:

Register name	Register type	Register application
I0..I99	Boolean	Physical digital inputs
O0..O99	Boolean	Physical digital outputs
D0..D19	Integer	Physical analog inputs
B0..B99	Integer	Volatile memory cells
M0..M99	Integer	Non-volatile memory cells
C0..C99	Integer	Auto decrement counters
S0..S99	Integer	Control, Status registers

Register name	Register type	Register application
A0..A99, V	String	String holders

2.5. Data Type Conversions

Conversions are performed between Boolean and Integer registers.

- Integer to Boolean Conversion- if *Integer* equals *not 0* then *true* else *false*, in other words: if *Integer* equals *0* then *false* else *true*.
- Boolean to Integer Conversion - *true* equals 1, *false* equals 0.

2.6. Operators

WESCL has the following types of operators:

- Assignment Operator

Operator	Description	Example
=	An assignment operator assigns the value to the left operand.	M1 = M2

- Comparison Operators

Operator	Description	Example
Equal (==)	Returns true if the operands are equal.	M1 == M2

Operator	Description	Example
Not equal (!=)	Returns true if the operands are not equal.	M1 != M2
Greater than (>)	Returns true if the left operand is greater than the right operand.	M1 > M2
Greater than or equal (>=)	Returns true if the left operand is greater than or equals the right operand.	M1 >= M2
Less than (<)	Returns true if the left operand is less than the right operand.	M1 < M2
Less than or equal (<=)	Returns true if the left operand is less than or equals the right operand.	M1 <= M2

- Arithmetic Operators

Operator	Description	Example
Multiplication (*)	Returns the product of two operands.	M1 * M2
Division (/)	Returns the quotient of two operands.	M1 / M2
Modulus (%)	Returns the integer remainder of dividing two operands.	M1 % M2
Addition (+)	Returns the sum of two operands.	M1 + M2
Subtraction (-)	Returns difference between two operands.	M1 - M2
Unary minus operator (-)	Returns the negation of its operand.	-M1

- Logical Operators

Operator	Description	Example
Logical AND (&&)	Returns *true if both operands are true; otherwise, returns false.*	operand1 && operand2
Logical OR (||)	Returns *true if either operand is true; if both are false, returns false.*	operand1 || operand2
Logical NOT (!)	Returns *false if its single operand can be converted to true; otherwise, returns true.*	!operand

- Special Operators

Operator	Description	Example
Comma operator (,)	Separator of multiple assignments in the action part of an instruction.	M1 = M2, M2 = M3, M3 = M4
Group ()	Parentheses allow overriding operator precedence. Up to 5 levels of nesting is allowed.	(M1 + M2) * (M3 - M4)

2.6.1. Operator Precedence

The precedence of operators determines their priorities when evaluating an expression.

The following table shows the precedence of operators, from the highest to the lowest.

Operator type	Individual operators

Operator type	Individual operators
group	( )
negation	! -
multiply	*
divide	/
modulus	%
addition	+
subtraction	-
relational	< <= > >=
equality	== !=
logical-and	&&
logical-or	||
assignment	=
comma	,

The operators of the same precedence level are resolved in the succession of their appearance in the expression.

2.7. Instructions

The WESCL program consists of instructions. Every instruction has an *expression* and an *action* part. The first one is ended by a colon and the second one is ended by a semicolon. The instruction syntax looks as follows:

```
expression : action ;
```

2.7.1. Expressions

An expression is any valid set of values, variables, operators, and expressions that evaluate to a single integer value. In every expression WESCL allows to use:

- Values: true, false, -32768..32767
- Variables: I0..I99, O0..O99, D0..D19, B0..B99, M0..M99, S0..S99, C0..C99
- Operators: (), &&, ||, !, - *(unary)*, *, /, %, +, -, >, <, >=, <=, !=, ==

Example 2-1. Expression examples

```
true
I5 && D0
M4 * (2 + !B99)
```

2.7.2. Actions

Conceptually, there are two types of *actions*: those that assign a value of expression to a variable (assignment action type), and those that simply represent a *variable* name (variable name action type).

One can use one of the following register identifiers as a variable: I0..I99, O0..O99, D0..D19, B0..B99, M0..M99, S0..S99, C0..C99.

Example 2-2. Assignment action type

```
O1 = I1
M5 = D0 * 2
B0 = M4 * (2 + !B99)
```

Example 2-3. A variable name action type

```
B0
-M0
I3
!I3
```

An *unary minus* operator or *not* operator can be inserted in front of *Variable Name Action*.

Additionally, any actions can be grouped in multi-actions using the *comma operator*.

Example 2-4. Multi action examples

```
/* assignment actions */
O1 = I1, M5 = D0 * 2, B0 = M4 * (2 + !B99)
B4, M0
O1 = I1, M5 = D0 * 2, B0 = M4 * (2 + !B99), B4, M0
```

2.7.2.1. String assignment actions

WESCL supports also action that can make assignment of string or copy string from one string-holder variable to another.

Example 2-5. String actions examples

```
A0 = "Hello!"
A1 = A0
V  = "index.html"
A2 = ""
```

2.8. Working with instructions

For better understanding how instructions work look at the conversion scheme below, using the well-known *C-language* equivalent:

Example 2-6. Instruction with a single assignment action

```
 /* WESCL */
 expression : action;

 /* real example in WESCL */
 2 * D0 + B0 : M1 = B0 * 3;




 /* C-language equivalent */
 if (expression)
     action;

 /* real example in C */
 if (2 * D0 + B0)
     M1 = B0 * 3;
```

Example 2-7. Instruction with multi action

```
 /* WESCL */
 expression : assignment_action1, assign-
ment_action2, assignment_action3;

 /* real example in WESCL */
 (I1 && I2) || B0 : M0 = 1, M1 = 5, M2 = M2 + 1;
```

```
/* C-language equivalent */
if (expression)
    assignment_action1, assignment_action2, assignment_action3;

/* real example in C */
if ((I1 && I2) || B0)
    M0 = 1, M1 = 5, M2 = M2 + 1;
```

Example 2-8. Variable name action

```
/* WESCL */
expression : var_name;

/* real example in WESCL */
B0 + B1 * D2 : M0;
```

```
/* C-language equivalent */
var_name = expression;

/* real example in C */
M0 = B0 + B1 * D2;
```

Example 2-9. Instruction with a multi variable name action

```
/* WESCL */
expression : var_name1, var_name2, var_name3;

/* real example in WESCL */
D0 : B0, D1, B1;
```

```
/* C-language equivalent */
int tmp = expression;

var_name1 = tmp;
var_name2 = tmp;
var_name3 = tmp;

/* real example in C */
int tmp = D0;

B0 = tmp;
D1 = tmp;
B1 = tmp;
```

Example 2-10. Instruction with a mixed mode action

```
/* WESCL */
expression : var_name1, assignment_action2, var_name2;

/* real example in WESCL */
D0 : B0, D0 = D0 + 1, B1;
```

```
/* C-language equivalent */
int tmp = expression;

var_name1 = tmp;

if (tmp)
    assignment_action2;
```

```
var_name2 = tmp;

/* real example in C */
int tmp = D0;

B0 = tmp;

if (tmp)
    D0 = D0 + 1;

B1 = tmp;
```

Example 2-11. Instruction with a complex mixed mode action

```
/* WESCL */
expression : -var_name1, assignment_action1, !var_name2;

/* real example in WESCL */
!(M0 + M2 * M3) : -B0, M4 = I1, !B5;
```

```
/* C-language equivalent */
int tmp = expression;

var_name1 = -tmp;

if (tmp)
    assignment_action1;

var_name2 = !tmp;

/* real example in C */
int tmp = !(M0 + M2 * M3);
```

```
B0 = -tmp;

if (tmp)
    M4 = I1;

B5 = !tmp;
```

Example 2-12. Instruction with a string assignment action

```
/* WESCL */
expression : assignment_action1, var_name2, assignment_action2;

/* real example in WESCL */
!(M0 + M2 * M3) : A0="Hello!\n", -B0, A1 = A0;
```

```
/* C-language equivalent */
int tmp = expression;

if (tmp)
    assignment_action1;

var_name2 = tmp;

if (tmp)
    assignment_action2;

/* real example in C */
char *A0;
char *A1;
int tmp = !(M0 + M2 * M3);

if (tmp)
    A0 = "Hello!\n";
```

```
B0 = -tmp;

if (tmp)
    A1 = A0;
```

2.9. Aliases

Some of WESCL language keywords have their aliases, which are treated identically as their corresponding original names. The following table shows all available aliases:

Description	Original keyword	Alias	Example
Logical NOT	!	not	not I1 : O3;
Logical AND	&&	and	I1 and I2 : O3;
Logical OR	||	or	I1 or I2 : O3;
Boolean value	true	ON	I1 or I2 : O3 = ON;
Boolean value	false	OFF	I1 or I2 : O3 = OFF;
Separator between expression and action part of instruction	:	==>	I1 or I2 ==> O3;

Chapter 3. Special registers

3.1. Counter registers

Registers C0..C99 work as auto-decremented counters. Each counter, containing the value greater than zero is decrement by one at the end of each WESCL loop, otherwise the value of a counter is unchanged.

Example 3-1. Using counters

```
     C8 : O8;
 not C8 : C8 = 2;
```

Generates a square wave on output O8 with a form factor 50%.

Attention, rewriting the above program in a following way:

```
 not C8 : C8 = 2;
     C8 : O8; /* counter C8 is al-
ways greater than zero at this moment! */
```

causes permanent setting the output O8 to logical 1.

3.2. Status registers

3.2.1. First loop indicator

The S0 register indicates execution of the first loop of WESCL firmware. It happens always after:

- Power on operation

- Software or hardware reset of WebEmbed
- Recompilation of `auto.scl` file. Initiated by the user.

Example 3-2. Using first loop indicator

```
S0 : M0 = 0, M1 = 0, M2 = 0;
```

initializing M0..M2 variables to zero at startup.

3.2.2. Date and time registers

The following table shows description of registers, which contain information about current date and time.

Register name	Description
S20	year (long format e.g. 2000)
S21	year (short format e.g. 00)
S22	month (1..12)
S23	day (1..31)
S24	weekday (0..6) Sunday = 0
S25	hour (0..23)
S26	minutes (0..59)
S27	seconds (0..59)
S28	milliseconds (0, 100, 200, 300, 400, 500, 600, 700, 800, 900)

3.2.3. Hardware counter

The following table shows registers, which are available on EvalBoard II only *(and*

operate on hardware counter connected to its digital input I12):

Register name	Description
S12	Shows the current value of hardware counter (0..255)
S13	Writing a non-zero value to this register causes resetting of the hardware counter.

3.2.4. Division by zero

S2 register stores information about division (/,%) by zero exception. In this situation S2 contains a non-zero value and the result of division is equal to -32768 or 32767 depending on the sign of the operation. WESCL does not clear this register automatically, so it is the programmer’s responsibility to do so.

Chapter 4. Interaction between WESCL and HTML-browser

The internal cgi-bin program called `wescl` makes it possible to interchange data between WESCL and HTML-browser.

4.1. HTML-browser to WESCL communication

All WESCL registers can be remote-controlled by HTML browser using the invocation of the cgi-bin wescl with parameters (the parameter's syntax must conform to the RFC 1738):

```
wescl?register_name=variable
```

or

```
wescl?register_name1=variable1&register_name2=variable2
```

Example 4-1. Setting register by HTML-browser

```
http://webembed/wescl?M0=123&B1=-456&A0=Hello%20I'm%20WESCL!
```

M0 is set to 123, B1 is set to -456 and "Hello I'm WESCL!" string is assigned to A0.

4.2. WESCL to HTML-browser communication

To present the state of any WESCL-register on the HTML-page, a special *&IOString(regname)* has to be inserted in the p-HTML-file.

Before the WebEmbed sends a HTML-page to the browser, the wescl parsing machine analyzes the user-defined "p-HTML" file, searching for &IOString(regname) placeholders and replacing them with the current value of the indicated register.

The example below shows how to place the current state of M0 and B1 register in the p-HTML file:

Example 4-2. Reading WESCL variables

```
<html>
  <body>
    <br>M0=&IOString(M0)
    <br>B1=&IOString(B1)
  </body>
</html>
```

To see the result, put this code to `abc.htm` file and point browser to URL: http://webembed/wescl?V=abc.htm.

Note: Notice that there is a difference in appearance (on the browser screen) between the page http://webembed/wescl?V=abc.htm generated by the cgi-bin wescl, and the page http://webembed/abc.htm generated out of the plain p-HTML file (not processed by the cgi-bin wescl).

4.2.1. V register

V register specifies the visualization p-HTML file, which is to be processed and sent to the browser by the cgi-bin wescl.

Example 4-3. Using V register

```
<html>
  <body>
    This is abc.html file.
```

```
  </body>
</html>
```

To run the example, place the above code in `abc.html` file and point browser to URL: http://webembed/wescl?V=abc.html

Note: After WebEmbed reset, the default value of the V register is "default.htm".

4.2.2. E register

The special read-only E register contains current information about WESCL state, such as status or error description generated by the WESCL compiler.

Example 4-4. Using E register

```
<html>
  <body>
    WESCL status: &IOString(E).
  </body>
</html>
```

To run the example, place the above code in `status.html` file and point browser to URL: http://webembed/wescl?V=status.html

Note: If everything is ok, the default value of E register is "Working".

Contents of the E register	Description
Working	Everything is ok.
ERROR: can not open /home/auto.scl file.	The auto.scl file doesn’t exists.
ERROR: while reading /home/auto.scl file.	Read error.

Contents of the E register	Description
ERROR: file /home/auto.scl is too large (>=8192 bytes).	Internal limit exceeded.
ERROR: Invalid Token Error	Source-code error
ERROR: Syntax Error	Source-code error
ERROR: Use simplier construction at line(...)	Too complicated expression (too many parentheses).
ERROR: Out of memory. Program too long.	Internal limit exceeded.

4.3. Compile WESCL programs

Compilation process is activated in the three following cases:

- Power on operation
- Software or hardware reset of WebEmbed
- Setting S0 register to non-zero value e.g. using following method: http://webembed/wescl?S0=1

Chapter 5. WESCL Communication Agent (WCA)

WESCL Communication Agent (WCA) provides ability to:

- Storage data to the circular buffer with a current timestamp;
- Compose e-mail messages;
- Include content of any WESCL-register, using &IOString() scheme, to the sending messages;
- Send e-mail messages;
- Send e-mail messages containing data from the circular buffer.

5.1. Circular Buffer

This is the implementation of a *Circular Buffer*, which is capable to store up to the 8 WESCL-registers along with the current timestamp into one record. All information of the buffer resides in the non-volatile memory. The following operations are provided to operate the buffer:

Operation	Description
Write a new record to the circular buffer	Setting a non-zero value to the S40 register causes copying the values of the S50..S57 registers to the circular buffer record, with the current timestamp automatically added.
Clear contents of the circular buffer	Setting a non-zero value to the S41 register removes all the entries from the circular buffer.

Operation	Description
Read the circular buffer length (capacity)	S47 register contains information about length *(capacity) of the circular buffer.*
Read the usage level of the buffer	S46 register contains information about the current usage level of the circular buffer (from 0 to the value of S47 register).
Read the circular buffer overflow status	If writing a new record to the buffer reaches the maximum of the buffer length, then S48 register contains a non-zero value. Thus, new record overwrites the old one. WCA does not clear S48 register automatically, so it is the programmer's responsibility to do so.

5.2. Message Structure

Message is composed of 3 parts:

- Message address;
- Message body;
- Message attachment.

The *message address* is mandatory. The *message body* and *message attachment* can be independently selected and added to the content of the message.

5.2.1. Message Address

The *message address* is a plain text file placed in the */home* directory. Valid names for *message address* are *address.number* where the number is in range 1..32768 (e.g.

address.13). The specified information must be placed within the first line! The format is similar to the *mailto:* scheme (RFC 2368). Recognized attributions are as following:

- from
- to
- CC
- Reply-To
- Subject

Example:

```
mailto:Person <somebody@dot.com>?subject=test&Reply-
To=Me <me@mydot.com>&from=<WebEmbed@dot.com>
```

> **Note:** Size of *message address* file is limited to 512 bytes. Larger files are internally truncated by WCA.

5.2.2. Message Body

The *message body* is a plain text file placed in the */home* directory. Valid names for *message body* are *body.number* where the number is in range 1..32768 (e.g. body.123). The *message body* file can also contain the placeholders of type &IOString(regname) to present the current state of any WESCL-register in the e-mail. It works in the same manner as sending a p-HTML file to the Browser through the WESCL-machine (see also: WESCL to HTML-browser communication).

> **Note:** Size of *message body* file is limited only by the capacity of the File System.

5.2.3. Message Attachment

The *message attachment* is a plain text file placed in the */home* directory. Valid names for *message attachment* are *attachment.number* where the number is in range 1..32768 (e.g. body.123). The *message attachment* file contains templates, how to format data on the circular buffer before sending it over e-mail.

There is one template for timestamp *%t*. This template is optional, but if exists, it must be placed at the first position. Up to 8 templates for each registers stored in the record can be placed next. Any combination of %d, %u, %x or %X can be specified (the usage of the formatting tags is the same as for printf() ANSI C library function).

Message attachment examples:

with timestamp and 3 formatted registers:

```
%t: %d %002d %u
```

with timestamp and 8 formatted registers:

```
%t.%d: %d,%d,%d,%d,%d,%d,%d
```

without timestamp (1 formatted register):

```
%05u
```

> **Note:** Size of *message attachment* file is limited to 512 bytes. Larger files are internally truncated by WCA.

5.3. Sending Messages

The following operations are provided to service messages:

Operation	Description
Sending message	Setting a non-zero value to the S42 register causes sending a message. Only 1 message can be sent at a time.
Status of sending message	If the message sending process is in progress, then S44 register contains a non-zero value.
Aborting sending message	Writing a non-zero value to the S43 register during sending a message process, causes aborting this operation (not immediately - to be sure check the S44 register).
Sending message error code	After sending message has completed, the major and minor return codes of this operation are stored into the S45 and S49 registers respectively.
Specifyng a message address (mandatory argument)	The S30 register specifies, which *address file should be used. Value stored in the S30 register is treated as an extension to the address file (e.g. if S30=5 then file address.5 will be used). Valid values are 1..32768.*
Specifying a message body	The S31 register specifies, which *body file should be used. Value stored in the S31 register is treated as an extension to the body file (e.g. if S31=2 then file body.2 will be used). Valid values are 0..32768, where 0 doesn't include body into the sent message.*

Operation	Description
Specifying a message attachment	The S32 register specifies, which *attachment file should be used. Value stored in the S32 register is treated as an extension to the attachment file (e.g. if S32=3 then file attachment.3 will be used). Valid values are 0..32768, where 0 doesn't include attachment into the sent message.*
Specifying a number of records	The S33 register specifies, how many records should be attached to the sent message. Values from 1 to the max. length of circular buffer specify the number of records. A zero causes sending all records. Register S33 has its meaning only if S32 contains a non-zero value. In case of the specified number of records being greater than the number of records available on the buffer, WCA will wait until the required number of records will be written to the buffer.

This example below shows how to send a message, when both input I1 and input I2 are equal true.

Example 5-1. Sending simple message

```
/* trigger */
I1 && I2 ==> B0;

/* check trigger and status
 * of completing previous sending action
 */
B0 && !S44 ==>
```

```
/* specify the filename ("/home/address.1")
 * for template of address
 */
S30 = 1,

/* optionally specify the filename
 * ("/home/body.3") for template of body
 */
S31 = 3,

/* don’t need attachment - it’s short message  */
S32 = 0,

/* send message */
S42 = 1;
```

Contents of the "/home/address.1" file

```
mailto:<tracelisting@dot.com>?from=Recorder <WebEmbed@dot.com>
```

Contents of the "/home/body.3" file

```
Gate opened.

 Current Date: &IOString(S22)/&IOString(S23)/&IOString(S21)
         Time: &IOString(S25):&IOString(S26):&IOString(S27)
```

This example shows how to send a message, containing the trace information for analog input D1 and D2.

Example 5-2. Sending complex message

```
/* record value of D1 at every minute */
 not S27 :
/* write D1 to the first argument of record */
```

```
S50 = D1,

/* write D2 to the second argument of record */
S51 = D2,

/* write a new record to the circular buffer */
S40 = 1;


/* check usage level of buffer, if more than 50% */
 S46 > (S47 / 2) ==> B0;

/* send a message, if previous was finished only */
 B0 && !S44 :

/* specify the filename ("/home/address.10")
 * for template of address
 */
S30 = 10,

/* specify the filename ("/home/body.10")
 * for template of body
 */
S31 = 10,

/* specify the filename ("/home/attachment.10")
 * for template of body
 */
S32 = 10,

/* send all records */
S33 = 0,

/* send message */
S42 = 1;
```

Contents of the "/home/address.10" file

```
mailto:<security@dot.com>?from=Gate 1 <WebEmbed@dot.com>
```

Contents of the "/home/body.10" file

```
History of D1 and D2 analog inputs.

+-----------+----+----+
|      Time       |  D1  |  D2  |
+-----------+----+----+
```

Contents of the "/home/attachment.10" file

```
| %t | %4d | %4d |
```

Hypotetical e-mail output

```
From MAILER-DAEMON Sat Dec 16 16:49:09 2000
Date: Sat, 16 Dec 2000 16:49:09 +0100 (CET)
From: Gate 1 <WebEmbed@dot.com>
Subject:

History of D1 and D2 analog inputs.

+-----------+----+----+
|      Time       |  D1  |  D2  |
+-----------+----+----+
| Dec 16 13:01:00 | 1234 |    3 |
| Dec 16 13:02:00 |  234 |   23 |
| Dec 16 13:03:00 |   34 |  123 |
| Dec 16 13:04:00 |  123 | 1456 |
| Dec 16 13:05:00 | 3034 |  743 |
| Dec 16 13:06:00 |   54 | 2123 |
.
.
```

.

5.3.1. Return Codes

The following table shows description of returned major/minor error codes.

Major Error Code (S45)	Description	Hints
0	Message sent successfully.	
1	Transmission aborted on user request.	Result of setting S43 to a non-zero value.
2	Cannot open requested file.	Check (according to values stored into the S30, S31, S32 registers) if appropriate files exist.
3	Error while reading file.	File system is probably corrupted.
4	Syntax error	Check syntax in configuration files.
5	Communication error.	Check also minor error description.

Minor Error Code (S49)	Description
0	No error.
1	Syntax error in "From" address.
2	Syntax error in "To" address.
3	Syntax error in "CC" address.
4	Bad character in the "From" address.
5	Bad character in the "To" address.

Minor Error Code (S49)	Description
6	Bad character in the "CC" address.
7	Bad character in the "Subject" address.
8	Error opening socket.
9, 10, 13, 15, 17, 23	SMTP server returning error code.
12	Error while sending data (HELO) to the SMTP server.
14, 16, 18, 20, 22, 24, 301, 302	Socket communication error.
100	Error while sending data (field TO:) to the SMTP server.
101	SMTP server returning error code (filed TO:).
200	Error while reading data from SMTP server.
202	Internal buffer overflow while reading data from SMTP server.
300	Bad character in sending message.
400	Error while sending data (field CC:) to the SMTP server.

5.4. Assigned Registers

The following tables show the summary of all registers used by WCA:

5.4.1. Argument Registers

Register name	Attribute	Description
S30	read-write	Address number (1..32768)
S31	read-write	Body number (0..32768)
S32	read-write	Attachment number (0..32768)
S33	read-write	Number of attached records (0 to value of S47)

5.4.2. Control Registers

Register name	Attribute	Description
S40	write-only	Write record to the buffer
S41	write-only	Clear all the buffer
S42	write-only	Send the message
S43	write-only	Abort sending message

5.4.3. Status Registers

Register name	Attribute	Description
S44	read-only	Message sending indicator
S45	read-only	Return error code
S46	read-only	Circular buffer usage level
S47	read-only	Length (capacity) of circular buffer

Register name	Attribute	Description
S48	read-only	Circular buffer overflow
S49	read-only	Minor error code

5.4.4. Data Registers

Register name	Attribute	Description
S50-S57	read-write	Data registers 0..7

Appendix A. WESCL grammar

This appendix presents WESCL grammar description using Extended-BNF format taken from PCCTS Toolkit. For more information on PCCTS, see: *ANTLR Website* (http://www.antlr.org).

```
/**
wescl.grammar 2.07 (01.09.2000)
written by Damian Wrobel <dwrobel@ertel.com.pl>

    WESCL - language description (Extended-BNF)
-*/

/*
    EBNF - short description
    ----------------

    Rules: describes a portion of the input language and
           consist of a list of alternatives.

    A rule syntax:

        rule : alternative1
             | alternative2
             ;

    where each of alternatives can by references to
    next rules, subrules or tokens.

    Subrules:

                  Form       Example

    plain subrule (...)     (OUTPUT | MEMORY)
    zero-or-more  (...)*    (statement)* "@"
```

```
    optional      {...}     {NOT} input


    Tokens:

    Tokens are defined with #token. Token reference indicate the
    token must be matched on the input file.

    Examples:

    #token OR       "or"
    #token AND      "and"
    #token XOR      "xor"

    #token Eof "@" // predefined end-of-file token


    For more information see also: <http://www.antlr.org>
*/




wescl :
      ( statement )* "@"
    ;

statement :
      expression COLON actionlist SEMICOLON
    ;

actionlist :
      action ( COMMA action )*
    ;
```

```
action :
      string
    | assign
    | var_name
    ;

string :
      HTML ASSIGN ( STRING | HTML )
    ;

assign :
      name ASSIGN expression
    ;

name :
      ( VALUE | INPUT | ADINPUT | OUTPUT | VMEMORY | MEM-
ORY | STATUS | COUNTER )
    ;

input :
      ( name | TRUE_ | FALSE_ | BITVAL )
    ;

var_name :
      { NOT | MINUS } name
    ;

expr_list :
      ( BRA expression KET )
    | input
    ;

operand :
      { NOT | MINUS } expr_list
    ;

multiplication :
```

```
      operand ( MUL operand )*
    ;

division :
      multiplication ( DIV multiplication )*
    ;

modulus :
      division ( MOD division )*
    ;

plus :
      modulus ( PLUS modulus )*
    ;

minus :
      plus ( MINUS plus )*
    ;

relational :
      minus ( ( LT | GT | LE | GE ) minus )*
    ;

equality :
      relational ( ( NOT_EQUAL | EQUAL ) relational )*
    ;

and :
      equality ( AND equality )*
    ;

expression :
      and ( OR and )*
    ;
```

```
#token UNARY_MINUS
#token NOT      "not|!"

#token MUL      "\*"
#token DIV      "/"
#token MOD      "%"

#token PLUS     "\+"
#token MINUS    "\-"

#token LT       "<"
#token GT       ">"
#token LE       "<="
#token GE       ">="

#token NOT_EQUAL "xor|!="
#token EQUAL     "=="

#token AND      "and|&&"
#token OR       "or|\|\|"

#token TRUE_    "true"
#token FALSE_   "false"

#token BRA      "\("
#token KET      "\)"
#token ASSIGN   "="
#token COLON    "\:|==>"
#token SEMICOLON ";"
#token COMMA    ","

#token HTML     "V|(A[0-9]{[0-9]})"
#token VMEMORY  "B[0-9]{[0-9]}"
#token MEMORY   "M[0-9]{[0-9]}"
#token STATUS   "S[0-9]{[0-9]}"
#token COUNTER  "C[0-9]{[0-9]}"
#token INPUT    "I[0-9]{[0-9]}"
```

```
#token ADINPUT  "D[0-9]|D[0-1]{[0-9]}"
#token OUTPUT   "O[0-9]{[0-9]}"
#token BITVAL   "ON|OFF"
#token VALUE    "[0-9]{[0-9]}{[0-9]}{[0-9]}{[0-
9]}" /* formally: SHRT_MIN .. SHRT_MAX */


#token STRING   « full-formated C-String »
```