TR-5000 Series

程式語言

Test Research, Inc.

April, 2004

Overview

The TR-8000 series test language is high-level and similar to the C programming language. A test program can be primarily divided into three sections,

The header section consists of UUT part definition, pin assignments, global constant definitions, and global variable declarations,

The function definition section consists of table allocations and user-defined routine definitions,

The main execution section consists of a sequence of test language statements that defines the plan of a test to be performed. The beginning of this section is the sole entry point of a test being run.

Like C, all test language statements are primarily composed of routines, expressions, and control-flow commands that direct the order in which statements are performed. A minor difference is statements are categorized as two types, digital test statements and utility statements. A digital test statement (test step) exclusively consists of nail state specification routines forming digital test patterns, test-specific looping and branching commands, and fail flag routines used to identify fail types for further implementation Otherwise, statements are utilities for test fail examination, test data process, test plan control, etc.

Routines break large tasks with different functions into smaller ones, thus clarifying the whole program and easing the pain of making changes. A routine consisting of a sequence of digital test steps is a test block or sub-block and called only in the main program or within a test block. A routine utilized to perform computations other than digital tests is a subroutine and can be called within other subroutines or in the main program. The TR-8000 series test language also provides standard routines such as nail state specification routines, fail examination routines, table data manipulation routines, file access routines, and so on, for executing digital test patterns and processing test data and results in programmer-defined test blocks and utility subroutines.

Terminology

The following terms are used to indicate specifically syntactic meanings in syntax descriptions of entire manual.

- All **boldfaced** style words are reserved in the test language.
- The syntactic items surrounded by left and right brackets ([ ]) may be omitted or included only one time.
- All syntactic items delimited by slashes (/) can be applied to the syntax associated.
- The syntactic items, which are sentences enclosed by < and >, are a list of syntactically identical statements.
- The syntactic items, which are composed of hyphenated words, are terminal symbols. A terminal symbol could be an identifier, a constant value, or a literal.
- All symbols not listed above in syntax statements are delimiters in the test language.
- The test language is case-insensitive.

Test Program Structure

The skeletons of a test program are depicted as follows:

PROGRAM program-name **;**

[**PART** device-part-name **;**]

[**INPUT** / **OUTPUT** / **BIDIR** <pin specifications list>]

[**GROUP** <group assignments list>]

[**CONST** <constant definitions list>]

[**VAR** <variable declarations list>]

[<table declarations and definitions list>]

[<digital test block declarations and definitions list>]

[<non-test subroutine declarations and definitions list>]

MAIN

<statements list>

END .

Identifier

An identifier is a sequence of letters and digits of any length. The first character must be a letter, and the underscore "_" counts as a letter. Identifiers are case-insensitive, for example, `INTVAL` and `IntVal` are identical identifiers. The test language reserves the following identifiers for specific use as **Keywords**, and must not be used otherwise.

BIDIR	BINARYFILE	BLOCK	BLOCKSUB
BOM	BYTE	CHAR	CONST
DLY	DO	DOWNTO	ELSE
END	EXPECT	FL	FLM
FLOAT	FOR	G1	G2
G3	G4	G5	GOTO
GROUP	HIN	HLIM	IF
INPUT	INTEGER	JF	JP
LLIM	LON	LOOP	LPT
MAIN	MEAS	MODE	OFFSET
ON	OUTPUT	PART	PROGRAM
RPT	SUBROUTINE	TABLE	TABLEPTR
TEXTFILE	THEN	TO	VAR
WHILE			

The test language also reserves the following identifiers as **Standard Routines** (routines written by another high-level language instead of the test language and embedded in TR-8000 series system), and must not be used.

CLOSE	DATE	DG	DH
DL	DX	FAIL	FAILCLR
FLAGFAIL	FLAGTESTFAIL	KDOFF	KDON
LOADBYTE	LOADTABLE	MC	MD
MDLY	MJ	ML	MQ
MR	MV	OPEN	READ
READLN	RESULTTABLE	SAVEBYTE	SAVETABLE
SG	SH	SL	SX
UDLY	USETABLE	WRITE	WRITELN

A **user-defined identifier** is an identifier excluding keywords and standard routine names listed above.

Data Types

The test language supports three classes of data types; they are scalar, array, and file types, which are described as follows.

Scalar Type	**Description**
`CHAR`	A signed single byte usually used to hold one character in the local character set.
`BYTE`	An unsigned single byte.
`INTEGER`	A signed integer.
`DWORD`	An unsigned integer of size 4 bytes.
`FLOAT`	A signed floating point.

An **array** type is composed of a scalar type name followed by left and right brackets in which the array size is specified. For example, `INTEGER[10]` is an integer array of size 10. A **string** is an array of characters, for example, `CHAR[10]` is a character array capable of holding up to 10 characters in a string. The first element of an array is located at index 1. An array index can be any integer expression, which includes integer variables and integer constants. Accessing an element of an array is an array identifier followed by left and right brackets in which specific index is evaluated.

Example
`S` is a 12-character array and assigned with the string `’hello, world’`. Referencing to `S[1]` gets the character `’h’`, `S[2]` gets `’e’`, `S[3]` gets `’l’`, etc.

A **file** is a source or destination data stream associated with a disk or other peripheral. The test language provides two specific data types for recording whatever information is necessary to control the stream.

File Type	**Description**
`TEXTFILE`	A text stream containing a sequence of printable ASCII characters, including new line escape characters and an end-of-file character.
`BINARYFILE`	A binary stream containing a sequence of raw bytes, which are the data stored in memory.

Constants

An **integer constant** like `8001` is an `INTEGER`. The value of an integer can be specified in binary or hexadecimal instead of decimal. A leading `0B` or `0b` on an integer constant means binary; a leading `0H` or `0h` means hexadecimal. For example, decimal `8001` can be written as `0B1111101000001` or `0b1111101000001` in binary and `0H1F41` or `0h1f41` in hex.

A **floating-point constant** like `8001.1` is a `FLOAT`.

A **character constant** is a `CHAR`, written as one character within single quotes, such as `'T'`. The value of a character constant is the numeric value of the character in the character set of the machine. For example, in the ASCII character set the character constant `'8'` has the value 56, which is unrelated to the numeric value 8. Therefore, a character constant also counts as an integer and can participate in any integer operations. For example, the statement, `CH=INT+'0';`, converts the numeric values 0 ~ 9 to character constant values '0' ~ '9'.

A **string constant**, or string literal, is a sequence of one more characters surrounded by single quotes, as in `'TR-5001 ICT'`. The quotes are not part of the string, but serve only to delimit it. The escape sequence `\'` represents the single-quote character in a string, for example, the string constant

```
'\'TR-5001\' ICT'
```

is output as

```
'TR-5001' ICT
```

Constants are defined using the keyword `CONST` followed by a list of constant value assignments, as in the following format.

CONST <constant-value-assignment-list>

Field	**Description**
<constant-value-assignment-list>	A list of assignment statements delimited by semicolons.

Example

```
CONST
    FILENAME='test.out';
    STARTADDR=0H1000;
```

Variable Declarations

Variables are user-defined identifiers and must be declared before use. A declaration specifies a type, and contains a list of one or more variables of that type, as in the following format.

VAR <declaration-list>

Field	Description
<declaration-list>	A list of declarations, which are composed of list of variable identifiers followed by a colon then a type, delimited by semicolons.

Example

```
VAR
    LOWER,UPPER,STEP:INTEGER;
    C:CHAR;
    LINE:CHAR[1000];
```

Operators

Arithmetic Operators

The binary arithmetic operators are

+	addition
-	subtraction or unary negation
*	multiplication
/	division
%	modulus

Integer division truncates any fractional part. The modulus operator % cannot be applied to float-point operands. The binary + and – operators have the same precedence, which is lower than the precedence of *, /, and %, which is in turn lower than unary **-**. Arithmetic operators associate left to right.

Relational and Logical Operators

The relational operators are

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

They all have the same precedence. Just below them in precedence are the equality operators:

`=`	equal to
`<>`	not equal to

Relational operators have lower precedence than arithmetic operators, thus an expression like `I<LIM-1` is taken as `I<(LIM-1)`, as would be expected. Below equality operators in precedence are the logical operators:

`&&`	logical AND
`||`	logical OR

The precedence of && is higher than that of ||. Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is determined. For example, a conditional expression:

```
IF I<=LIM && C<>' ' THEN
    STR[I]=C;
```

The expression `I<=LIM` must be tested first and the next expression `C<>' '` stops testing as if this test fails.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, otherwise, 0 if the relation is false. Usually, such an expression is also called a Boolean expression.

The unary negation operator `!` logically inverts the result of a Boolean expression, that is, converts a non-zero operand into 0, and a zero operand into 1.

Bit-wise Operators

The bit-wise operators are

`&`	bit-wise AND
`|`	bit-wise OR
`^`	bit-wise exclusive OR (XOR)
`<<`	left shift
`>>`	right shift
`~`	bit-wise NOT (1's complement)

The bit-wise operators provide for bit level manipulation, and only can be applied to integral operands, that is, `CHAR`, `BYTE`, and `INTEGER`. The following example is to extract the high and low nibbles of a byte.

```
VAR
   BYTEVAL:BYTE;
   BYTESTR:CHAR[2];

SUBROUTINE EXTRACT;
{
   BYTESTR[1]=(BYTEVAL>>4)+'0';
   BYTESTR[2]=(BYTEVAL&0H0F)+'0';
};
```

Given that the `BYTEVAL` is the numeric value AA in hex, the `BYTESTR` is the string 'AA' after executing the `SUBROUTINE EXTRACT`.

Assignment Operator

The assignment operator is `=`, which is symbolically identical to the equality operator, but serves to different operations. The expression at the right-hand side of the assignment operator is first evaluated, and the value of this expression replaces that of the object referred to by the left-hand side variable. For example, `I=I+1`, given that the value of `I` is 1, the value of `I` would be 2 after evaluating the expression.

Expressions

An operational expression is a list of operands connected by operators, and is evaluated as a value in accordance with the precedence and associativity of operators applied. Only variables having scalar data types are valid operands in operational expressions due to scalar operators. A primary operational expression has single operand without any operators applied, such as, a constant, a string literal, and an identifier, are all primary expressions. The data type of a primary expression is identical to that of the operand associated. A parenthesized expression is an expression enclosed by left and right parentheses, having the highest precedence in evaluation. For example, the expression, `C=A+B*3`, and `C` is 7 given that `A` is 1 and `B` is 2, but `C` is 9 for the expression, `C=(A+B)*3`, due to higher precedence of parentheses than that of multiplication.

The data type of the value evaluated by an expression depends on the data type of operands involved. The types of operands should be converted to a common type as an operator has operands of different types. The type conversions follow the rules listed below.

The scalar types, array types, and file types cannot be converted to one another.
The `INTEGER`, `BYTE`, and `CHAR` can be freely converted.
Only `INTEGER` can be converted to `FLOAT`.
The `FLOAT` cannot be converted to `INTEGER`, `BYTE`, and `CHAR`.

A routine call is an expression called the routine name designated by `SUBROUTINE` or `BLOCK` or `BLOCKSUB`. Such an expression has single operand and returns no value, so no operators are applied.

Statements

An expression such as `X=100` or `I=I+1` or `PRINTRESULT` followed by a semicolon becomes a **statement**, as in

```
X=100;
I=I+1;
PRINTRESULT;
```

The semicolon is a statement terminator in the test language. Braces { and } are used to group statements together into a **compound statement**, which are syntactically equivalent to a single statement. The braces around multiple statements after an `IF THEN`, `ELSE`, `FOR TO DO`, `WHILE DO` are an obvious example when these statements have to be executed for that condition.

Analog Testing

The test language implements these statements to make analog components measurements and tests.

Test Language	Analog Measurement Functions
MR	Resistance (ohms)
MC	Capacitance (farads)
ML	Inductance (Henries)
MJ	Jumper (open/short)
MD	Diode (volts)
MQ	Transistor (volts)

In addition to the analog test statements that control the test devices, the test language also contains statements to have the user customize the test program. These test statements include flow control and data manipulation statements; refer to the *Utility Statements* for further details.

Analog tests can be exclusively executed within the component test steps.

Analog Test Statements

MR / MC / ML / MJ / MD / MQ

Type
Standard routine

Description
Use the MR, MC, ML, MJ, MD, and MQ statements to set up the analog instruments for analog devices measurements, store the results to user-defined variables, and test the results against limits. Refer to the *C-1 Analog Test Theory* for detailed information about analog instrumentation resources and analog test algorithms.

Syntax
[pass-fail **=**] **MR**/**MC**/**ML**/**MJ**/**MD**/**MQ** **(** <parameter list> **)** **;**

Field	**Description**
pass-fail	*Optional*. An integer variable used to store the test result. The variable pass-fail is set to 1 if a failure is detected; otherwise, the variable is set to 0 for passing the test.
<parameter list>	A list of parameter value assignments delimited by colons, referring to the format below. parameter-identifier **=** parameter-value **,** Allowable parameter identifiers are **PART**, **BOM**, **EXPECT**, **OFFSET**, **HLIM**, **LLIM**, **MODE**, **HIN**, **LON**, **DLY**, **G1**, **G2**, **G3**, **G4**, **G5**, **RPT**, and **MEAS**. PART, EXPECT, MODE, HIN, and LON are required for all types of statements. BOM is required in MD and MQ statements. All other parameters are optional.

Identifier	**Value**
PART	A string literal representing a device name listed in the component test steps or PINS.ASC file.
EXPECT	A string literal representing an expected value measured.

	BOM	A string literal required for MD and MQ statements representing the turn-on voltage applied to a diode or transistor; optional for the other statements representing a value listed in BOM.
	OFFSET	*Optional*. A floating-point value representing the compensation from a measured value. Default is 0.
	HLIM LLIM	*Optional*. An integer value representing the tolerance in percentage of an expected value. Default is –1 for ignoring the limits.
	MODE	Specifies an integer value to apply specific analog instrument to a DUT.
	HIN/LON	An integer value representing the source/sink nail of a DUT.
	DLY	*Optional*. An integer value representing the delay in milliseconds before starting to measure. The maximum value is 500ms with a default of 0.
	G1/G2/G3/ G4/G5	*Optional*. An integer value representing a guarding nail with a default of 0.
	RPT	*Optional*. An integer value representing the maximum retest times if a DUT fails the test.
	MEAS	*Optional*. Measurement result is stored in the floating-point variable specified by this parameter.

Refer to the *Program Development B-1-7 MDA Debug* for detailed descriptions about these parameters.

Example

```
MR(PART='R1',EXPECT='4',HLIM=8,LLIM=8,MODE=0,HIN=2,LON=1);
MR(PART='R3',EXPECT='100K',HLIM=3,LLIM=3,MODE=1,HIN=14,LON=1);
MC(PART='C5',EXPECT='10n',HLIM=8,LLIM=8,MODE=2,HIN=28,LON=29,
DLY=4);
MD(PART='ZD1',BOM='0.7V',EXPECT='0.75V',HLIM=10,LLIM=10,MODE=0
,HIN=49,LON=48);
MQ(PART='Q1-BCE',BOM='1.5V',EXPECT='0.095V',HLIM=10,MODE=3,
HIN=52,LON=53,G1=48);
```

Digital Testing

UUT Pin Definitions

A test language program used to test an IC should reference to one or more pins of that IC. This reference is made possible through a list of pin specifications in the following format:

pin-type-designator <pin ID assignment list>

Field	Description
pin-type-designator	Three keywords used to define the type of an IC pin: **INPUT**, **OUTPUT**, and **BIDIR**.
<pin ID assignment list>	A list of pin to number or pin to test nail assignment expressions delimited by semicolons, referring to the format below. pin-name **=** test-nail / pin-number **;** pin-name — User-defined identifier. All pins must be associated with a unique name. test-nail — A positive integer representing the number of digital resources connected to a pin. pin-number — A string literal representing the pin number.

Example
```
INPUT
    WE='A1';
    OE='B1';
    CE='C1';
    . . . . . . . .

OUTPUT
    STAT=1130;
    . . . . . . . .

BIDIR
    D0='A3';
    D1='11'; // 11 is a pin number instead of a test nail
    . . . . . . . .
```

Grouping pins with the same types facilitates the program coding and improves the program readability, specifically for bus pins. The test language supports the syntax for group definitions as follows:

GROUP <group pins assignment list>

Field	**Description**
<group pins assignment list>	A list of pins to group assignment expressions delimited by semicolons in the following format: group-name **= (** <pin name list> **) ;** group-name User-defined identifier. All groups must be associated with a unique name. <pin name list> A list of defined pin names delimited by commas. The pins listed from left to right correspond to the bit sequence from MSB to LSB.

Example

```
GROUP
    DATABUS=(D7,D6,D5,D4,D3,D2,D1,D0);
    ADBUS=(FWH3,FWH2,FWH1,FWH0);
```

Note that the maximum number of pins contained in a group is 32.

Nail State Specification Routines

The nail state specification routines listed below are used within BLOCK/BLOCKSUB to specify the drive/sense and logic states of nails.

DH	**DL**	**DX**	**DG**
SH	**SL**	**SX**	**SG**

The valid parameters for DH/DL/SH/SL are pin names or nail numbers or both. The only available parameters for DG/SG are group names designated by GROUP. The DX/SX can take them all as parameters. Use an asterisk as a unique parameter to indicate all pins specified as parameters. Commas are delimiters to separate individual parameters.

The detailed states of nails specified by above commands are described in the following table.

Drive States		
DH	Drive High	Enable driver and place nail in a high state
DL	Drive Low	Enable driver and place nail in a low state
DX	Drive Don't Care	Disable and place drivers in a high-impedance state
DG	Drive Group	Enable group of drivers and place related nails in specified states
Sense States		
SH	Sense High	Enable sensor and expected nail state is sensed high
SL	Sense Low	Enable sensor and expected nail state is sensed low
SX	Sense Don't Care	Ignore these sense states
SG	Sense Group	Enable group of sensors and expected nails states are sensed in specified states

Digital Test Keywords

Keywords listed below define a user-customized digital test block/sub-block. The implementation (a sequence of digital test statements) for a block/sub-block is delimited using left and right braces. Definitions are provided later in the manual.

BLOCK	BLOCKSUB

The following keyword is used to allocate a binary data segment and map it to a set of pins with drive/sense state assigned.

TABLE

Use the following keyword to declare a pointer to a table. Reference different segments of a table by the pointers to reduce the table count.

TABLEPTR

The control-flow commands designated by the following keywords are exclusively used within BLOCK/BLOCKSUB to specify the order in which digital test steps are executed.

JP	JF	
FL	FLM	LOOP

The limitations for these keywords are as follows:

The BLOCKSUB is called only in the BLOCK,
The TABLE/TABLEPTR can only be referenced in the BLOCK,
The BLOCK is called only in the main program.

Digital Test Statements

BLOCK

Type
Keyword

Description
The BLOCK keyword defines and executes a segment of a digital test or a complete one. Any valid digital test statements within the BLOCK delimited by left and right braces define each step taken by the driver/sensor resources. The drive states of nails are incremental within and across blocks. The sense states of nails imply the 'Don't-Care' state unless they are specifically mentioned at the statement as being SH/SL/SG.

Syntax
BLOCK label **;**
{
        <valid digital test statements>
} ;

Field	Description
label	User-defined identifier for the block. All blocks should be associated with a unique name.
<valid digital test statements>	Composed of any valid test command keywords, including table access pointers designated by TABLE/TABLEPTR and subroutine calls designated by BLOCKSUB.

Example
```
TABLE ADDRTAB : 0H20000;
{
   DH(AD17,AD16,. . .,AD2);
};

TABLE DATATAB : 0H80000;
{
   DH(D31,D30,. . .,AD17,AD16,. . .,D0);
};

TABLEPTR BANK1 = DATATAB;
. . . . . . . .
BLOCKSUB INIT;
{
   DL(*);
   . . . . . . . .
};

BLOCK PROGRAMMING;
{
   INIT; // call to sub-block and execute the test

   // enable bank1 and bank2
   DH(F1TST,F2TST);

   FL 2048
   {
      ADDRTAB+; // reference the table

      DH(F1ALE,F2ALE,PROG,XE);
      DL(F1ALE,F2ALE);
      DX(ADDRBUS);
      . . . . . . . .
      BANK1+; // reference the table
      DH(F1DLE);
      DL(F1DLE);
      . . . . . . . .
      DX(DATABUS);
      DH(YE);;;;;;;;;;;; // delay to program the data
      . . . . . . . .
      DL(YE);
      DL(PROG,XE);
      . . . . . . . .
   };

   // disable bank1 and bank2
   DL(F1TST,F2TST);
};
. . . . . . . .
MAIN
   . . . . . . . .
   PROGRAMMING; // call to block and execute the test
   . . . . . . . .
END.
```

BLOCKSUB

Type

Keyword

Description

The BLOCKSUB keyword defines and executes a segment of a digital test, which is common to different blocks. Any valid digital test statements in the BLOCKSUB delimited by left and right braces define each step taken by the driver/sensor resources. The drive states of nails are incremental within and across sub-blocks. The sense states of nails imply the 'Don't-Care' state unless they are specifically mentioned at the statement as being SH/SL/SG. Call to blocks or sub-blocks and reference to tables are not permitted in the BLOCKSUB.

Syntax

BLOCKSUB label ;

{

    <valid digital test statements>

} ;

Field	**Description**
label	User-defined identifier for the sub-block. All sub-blocks should be associated with a unique name.
<valid digital test statements>	Composed of any valid test command keywords, excluding table access pointers designated by TABLE/TABLEPTR and subroutine calls designated by BLOCK/BLOCKSUB.

Example

```
BLOCKSUB CHECKSTATUS;
{
    . . . . . . . . .
    DL(SCK) DH(SI);
    DH(SCK); DL(SCK);
    . . . . . . . . .
    FLM 10000
    {
        DH(SCK); DL(SCK);
        . . . . . . . . .
        SL(SO) FLAGFAIL(301); // 1: busy, 0:ready
        DL(SCK);
    };
    DH(CS);
};
```

DG / DH / DL / DX

Type

Standard routine

Description

Use these routines to form digital Drive test statements, in turn, being translated to digital test steps, which are separated by semicolons. Individual semicolons in a statement list represent test steps kept from the previous one. Take an asterisk as the parameter of DH/DL/DX to specify the states for all pins designated by INPUT and BIDIR.

Syntax

DH / **DL** / **DX** **(** <valid parameters> **)** … **;**

DG **(** group_name **=** value **,** … **)** … **;**

Field	**Description**
<valid parameters >	Pin names designated by INPUT, OUTPUT, BIDIR, or nail numbers. Individual parameters are delimited using commas. Group names designated by GROUP are permitted only for DX. Use an asterisk as the unique parameter.
group_name	User-defined identifier designated by GROUP
value	A positive integer or integer constant designated by CONST represents logic states.

Example

```
INPUT
   CTRL=1234;
   . . . . . . . .

OUTPUT
   STAT=1024;
   . . . . . . . .

BIDIR
   D3=1111;
   D2=1112;
   D1=1113;
   D0=1114;
   . . . . . . . .

GROUP
   DBUS=(D3,D2,D1,D0);
```

```
    . . . . . . . .
BLOCK TEST;
{
    DH(*);
    /*
        drive all pins designated by INPUT and BIDIR to high,
        that is, nail 1234 and nails 1111~1114 are driven high
    */

    DL(CTRL);;;;
    /*
        the pin CTRL is driven low and keeps driving low
        on the next three steps
    */

    DG(DBUS=0HA);
    /*
        the states of D3~D0 are driven
        high low high low (1010)
    */

    DX(DBUS); // drive D3~D0 to high-impedance
              // that is, to turn off the drivers D3~D0
    . . . . . . . .
};
```

The nail states through the block execution are shown as follows:

Nail State / Test Step	NAIL NUMBER					
	1234	1111	1112	1113	1114	1024
Step 1	1	1	1	1	1	X
Step 2	0	1	1	1	1	X
Step 3	0	1	1	1	1	X
Step 4	0	1	1	1	1	X
Step 5	0	1	1	1	1	X
Step 6	0	1	0	1	0	X
Step 7	0	X	X	X	X	X

Note:
1 stands for driving high, 0 stands for driving low, X stands for turning off the driver or ignoring the result.

FAILCLR

Type
Standard routine

Description
Use FAILCLR routine to clear all or the specified fail flags, which are set by FLAGFAIL routines within a BLOCK. All fail flags taken as parameters of FLAGFAIL routines through the whole program are initialized to clear automatically prior to the beginning of the main test designated by MAIN.

Syntax
FAILCLR ;
FAILCLR (<fail flag number list> **) ;**

Field	**Description**
<fail flag number list>	A list of positive integers and integer constants representing fail flags. Individual parameters are delimited using commas.

Example
```
MAIN
    FAILCLR;
    . . . . . . . .
    IF FAIL(101) THEN
    {
        FAILCLR(101);
        . . . . . . . .
    };
    . . . . . . . .
END.
```

FAIL

Type

Standard routine

Description

Use FAIL routine to return the pass/fail state of a numbered flag. 1 represents the pass state, and 0 represents the fail state.

Syntax

FAIL (n **)**

Field	**Description**
n	A positive integer or integer constant designated by CONST. 0 is reserved for system fail flag.

Example

```
MAIN
    . . . . . . . .
    IF FAIL(102) THEN
    {
       WRITELN(LPT,'CHECK ID FAILED');
       FLAGTESTFAIL(PROGRAMFAIL);
    };
    . . . . . . . .
END.
```

FL

Type

Keyword

Description

The FL keyword provides the loop function in a BLOCK/BLOCKSUB. The sequence of statements to be repeated is delimited using the left and right braces. A loop count indicates the maximum number of times a loop is executed. Other exit condition implied from the FL is “exit if failed”. The fail state at the end of the loop is determined as long as any statement in the sequence fails.

Syntax

FL count

{

<sequence of statements>

} ;

Field	Description
count	A positive integer or integer constant designated by CONST
<sequence of statements>	Any test statements based on the valid test keywords

Example

```
BLOCK TEST;
{
   . . . . . . . .
   FL 100
   {
      . . . . . . . .
      DH(CLK);
      SL(SO) FLAGFAIL(101);
      /*
         the fail flag 101 is set as SO is sensed high
         and the loop stops iterating at the end of the
         current execution.
      */

      DL(CLK);
      . . . . . . . .
   };
   . . . . . . . .
};
```

FLAGFAIL

Type

Standard routine

Description

The FLAGFAIL routine sets the flag numbered by n when related digital test statement fails. The FLAGFAIL can be appended to any individual test statement, which executes a digital test step, within a BLOCK. Examine the flag status to identify which test steps failed at BLOCK completion.

Syntax

FLAGFAIL (n **) ;**

Field	Description
n	A positive integer or integer constant designated by CONST. 0 is reserved for system fail flag.

Example

```
TABLE RDDATATAB : 0H80000
{
   SH(D8,D7,D6,D5,D4,D3,D2,D1,D0);
};

BLOCK TEST;
{
   . . . . . . . . .
   SL(STAT) FLAGFAIL(102);
   . . . . . . . . .
   RDDATATAB+ FLAGFAIL(103);
   . . . . . . . . .
};

MAIN
   . . . . . . . . .
   USETABLE(RDDATATAB);
   TEST; // execute the block

   IF FAIL(103) THEN
   {
      WRITELN('VERIFY FAILED');
      . . . . . . . . .
   }
   . . . . . . . . .
END.
```

FLAGTESTFAIL

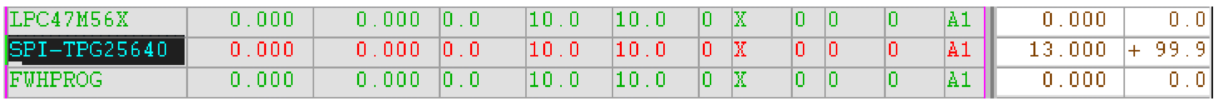
Type
Standard routine

Description
The FLAGTESTFAIL routine fails the test statement without requiring a hardware measurement. Without FLAGTESTFAIL there is no way to set a failure for evaluating variables.

Syntax
FLAGTESTFAIL ;

Example
```
MAIN
    . . . . . . . .
    IF FAIL(100) THEN
        FLAGTESTFAIL;
    . . . . . . . .
END.
```

The test step shown below is the result after executing the IF statement.



LPC47M56X	0.000	0.000	0.0	10.0	10.0	0	X	0	0	0	A1	0.000	0.0
SPI-TPG25640	0.000	0.000	0.0	10.0	10.0	0	X	0	0	0	A1	13.000	+ 99.9
FWHPROG	0.000	0.000	0.0	10.0	10.0	0	X	0	0	0	A1	0.000	0.0

FLM

Type

Keyword

Description

The FLM keyword provides the loop function in a BLOCK/BLOCKSUB. The sequence of statements to be repeated is delimited using left and right braces. A loop count indicates the maximum number of times a loop is executed. Other exit condition implied from the FLM is "exit if passed". The fail state at the end of the loop is determined as long as any statement in the sequence fails.

Syntax

FLM count

{

    <sequence of statements>

} ;

Field	**Description**
count	A positive integer or integer constant designated by CONST
<sequence of statements>	Any test statements based on the valid test keywords

Example

```
BLOCK TEST;
{
   . . . . . . . .
   FLM 100
   {
      . . . . . . . .
      DH(CLK);
      SL(SO) FLAGFAIL(101);
      /*
        the loop stops iterating as SO is sensed low
        the fail flag 101 will be cleared while repeating
        the execution (SO is sensed high)
      */

      DL(CLK);
      . . . . . . . .
   };
   . . . . . . . .
};
```

JF

Type

Keyword

Description

The JF keyword transfers the program execution sequence to the specified label when the related test statement fails within a BLOCK/BLOCKSUB. The scope of the label must be in the same level as the one of JF, that is, jumping into, out of, or between BLOCK/BLOCKSUB sections are not permitted. Neither does jump into, out of, or between loop statements.

Syntax

JF label ;

Field	Description
label	User-defined identifier on the statement to which the program transfers

Example

```
BLOCK TEST;
{
    . . . . . . . .
    FL 100
    {
        . . . . . . . .
        SG(HIBYTE=0HAA) JF ENDVERIFY;
        . . . . . . . .
    ENDVERIFY:
        . . . . . . . .
    };
    . . . . . . . .
};
```

JP

Type
Keyword

Description
The JP keyword transfers the program execution sequence to the specified label when the related test statement passes within a BLOCK/BLOCKSUB. The scope of the label must be in the same level as the one of JP, that is, jumping into, out of, or between BLOCK/BLOCKSUB sections are not permitted. Neither does jump into, out of, or between loop statements.

Syntax
JP label **;**

Field	Description
label	User-defined identifier on the statement to which the program transfers

Example
```
BLOCK TEST;
{
    . . . . . . . .
    SG(DBUS=0H6D) JP ENDTEST;
    . . . . . . . .
ENDTEST:
    . . . . . . . .
};
```

KDOFF / KDON

Type
Standard routine

Description
Use KDOFF and KDON routines to close and open the specified digital relays of pins. All digital relays of pins designated by INPUT, OUTPUT, and BIDIR are initialized to close automatically prior to the beginning of the main test, and reset to open at the end of the test.

Syntax
KDOFF / **KDON (** <valid parameters> **) ;**

Field	Description
<valid parameters>	Pin names designated by INPUT, OUTPUT, BIDIR, or nail numbers. Individual parameters are delimited using commas.

Example
```
MAIN
   KDON(F1ALE,F2ALE,. . .,XE);
   . . . . . . . .
   KDOFF(F1ALE,F2ALE,. . .,XE);
END.
```

LOADTABLE

Type

Standard routine

Description

The LOADTABLE routine loads data from a binary file or a table file used in on-board flash programming to an allocated memory in which logic states (test patterns) of the specified drive/sense pins reside. This permits On-The-Fly programming for various items like a board serial number or changing the contents of a programmable device without creating a new test program.

Syntax

LOADTABLE (table-name **,** data-file **) ;**

Field	**Description**
table-name	User-defined identifier designated by TABLE/TABLEPTR
data-file	A string literal or string constant designated by CONST represents the file name of a binary or TAB file.

Example

```
CONST
    DATAFILENAME='data.bin';
    . . . . . . . .

TABLE DATATAB : 0H4;
{
    SH(D8,D7,D6,D5,D4,D3,D2,D1,D0);
};
. . . . . . . .

MAIN
    . . . . . . . .
    LOADTABLE(DATATAB,DATAFILENAME);
    USETABLE(DATATAB);
    . . . . . . . .
END.
```

After loading data from the binary file DATAFILENAME to the table DATATAB, actual data calculated while calling USETABLE in the table is shown below.

Data in the binary file:

```
00000000h: 43 6F 70 79 . . . . . . . .
. . . . . . . .
```

Data representing the test patterns and steps in the table:

Table Data / Test Step	Bit 7 D7	Bit 6 D6	Bit 5 D5	Bit 4 D4	Bit 3 D3	Bit 2 D2	Bit 1 D1	Bit 0 D0
Step 1	L	H	L	L	L	L	H	H
Step 2	L	H	H	L	H	H	H	H
Step 3	L	H	H	H	L	L	L	L
Step 4	L	H	H	H	H	L	L	H

Notes:

1. L stands for expected state is sensed low, H stands for expected state is sensed high.
2. Each table reference executes one step, for example, first occurrence of the table reference runs the step 1, that is, comparing the data read from UUT with the expected pattern “01000011”.
3. The plus symbol appended to the table reference, for example, DATATAB+, increases the table pointer to the next step after current execution. The table pointer stops at the last step for further increments.
4. The minus symbol appended to the table reference, for example, DATATAB-, decreases the table pointer to the prior step after current execution. The table pointer stops at the first step for further decrements.
5. The table reference statement cannot be placed in the same statement as any nail state specifications, control-flow commands, and any other table references. Only FLAGFAIL routine can be used in the same statement with the table reference.
6. The appropriate USETABLE or RESULTTABLE must be called somewhere in the main program before referencing the table.

LOOP

Type

Keyword

Description

The LOOP keyword provides the unconditional loop function in a BLOCK/BLOCKSUB. The sequence of statements to be repeated is delimited using the left and right braces. A loop count indicates the maximum number of times a loop is executed. The pass or fail state of the loop is determined at the end of the loop. A failure occurs if any statement in the sequence fails.

Syntax

LOOP count

{

<sequence of statements>

} ;

Field	Description
count	A positive integer or integer constant designated by CONST
<sequence of statements>	Any test statements based on the valid test keywords

Example

```
BLOCK TEST;
{
    . . . . . . . .
    LOOP 100
    {
        . . . . . . . .
        DH(CLK);
        SL(SO) FLAGFAIL(101);
        /*
            the fail flag 101 is set as SO is sensed high
            however, the loop continues iterating until the
            loop count expires
        */

        DL(CLK);
        . . . . . . . .
    };
    . . . . . . . .
};
```

RESULTTABLE

Type
Standard routine

Description
The RESULTTABLE routine loads a table pointer designated by TABLE/TABLEPTR to the first test step of a TABLE. This also permits loading a table pointer to the specified test step offset from the first one. Use the RESULTTABLE statement accompanied with table reference statements to write result patterns read from UUT into a TABLE. The RESULTTABLE can only be created with sense pins.

Syntax
RESULTTABLE (table-name [**,** step-offset] **) ;**

Field	**Description**
table-name	User-defined identifier designated by TABLE/TABLEPTR
step-offset	*Optional* parameter. An integer or integer constant designated by CONST represents the test step offset from the first step (the beginning of a TABLE).

Example
```
TABLE DATATAB : 0H40000;
{
    SH(D8,D7,D6,D5,D4,D3,D2,D1,D0);
};
. . . . . . . .

MAIN
    . . . . . . . .
    RESULTTABLE(DATATAB);
    . . . . . . . .
    SAVETABLE(DATATAB,'outdata.bin');
    . . . . . . . .
END.
```

SAVETABLE

Type
Standard routine

Description
The SAVETABLE routine writes RESULTTABLE data to a binary file for further manipulation. Only sense data resides in a RESULTTABLE.

Syntax
SAVETABLE (table-name **,** data-file **) ;**

Field	Description
table-name	User-defined identifier designated by TABLE/TABLEPTR
data-file	A string literal or string constant designated by CONST represents a binary file name.

Example
```
CONST
    OUTFILENAME='outdata.bin';
    . . . . . . . .

TABLE DATATAB : 0H40000;
{
    SH(D8,D7,D6,D5,D4,D3,D2,D1,D0);
};
. . . . . . . .

MAIN
    . . . . . . . .
    RESULTTABLE(DATATAB);
    . . . . . . . .
    SAVETABLE(DATATAB,OUTFILENAME);
    . . . . . . . .
END.
```

SG / SH / SL / SX

Type

Standard routine

Description

Use these routines to form digital Sense test statements, in turn, being translated to digital test steps, which are separated by semicolons. Individual semicolons in a statement list represent test steps kept from the previous one. The nail states are not incremental and go to Don't-Care (X) state.

Syntax

SH / **SL** / **SX** **(** <valid parameters> **)** … **;**

SG **(** group-name **=** value **,** … **)** … **;**

Field	Description
<valid parameters>	Pin names designated by INPUT, OUTPUT, BIDIR, or nail numbers. Individual parameters are delimited using commas. Group names designated by GROUP are permitted only for DX. Use an asterisk as the unique parameter.
group-name	User-defined identifier designated by GROUP
value	A positive integer or integer constant designated by CONST represents expected logic states.

Example

```
INPUT
    CLK=1024;
    . . . . . . . .

BIDIR
    D3=1111;
    D2=1112;
    D1=1113;
    D0=1114;
    . . . . . . . .

GROUP
    DBUS=(D3,D2,D1,D0);
    . . . . . . . .

BLOCK TEST;
{
    DL(CLK,D3,D2,D1) DH(D0);
    DH(CLK);
    DL(CLK) DX(D3,D2,D1,D0);
    DH(CLK) SG(DBUS=0H5);
    DL(CLK);
    DH(CLK) SL(D3,D1) SH(D2,D0);
    DX(CLK);
};
```

The nail states through the block execution are shown as follows:

Nail State / Test Step	NAIL NUMBER				
	1024 (CLK)	1111 (D3)	1112 (D2)	1113 (D1)	1114 (D0)
Step 1	0	0	0	0	1
Step 2	1	0	0	0	1
Step 3	0	X	X	X	X
Step 4	1	L	H	L	H
Step 5	0	X	X	X	X
Step 6	1	L	H	L	H
Step 7	X	X	X	X	X

TABLE

Type
Keyword

Description
Use the TABLE keyword to allocate a binary data segment representing digital test steps of each in turn being mapped to a set of pins with drive/sense state specified. The TABLE can be only referenced by the TABLE name (implied a pointer to the TABLE) or the pointers designated by TABLEPTR within a BLOCK. A plus appended to a table pointer advances a pointer to the next test step after the current test step completion. A minus appended to a table pointer decreases a pointer to the previous test step after the current test step completion. The first pin listed is used as the most significant bit (MSB), and the last as the least significant bit (LSB), in mapping to data loaded from or saved to a binary file.

Syntax

TABLE name **:** size **;**
{
        DH / **SH** **(** <pin map list> **) ;**
} ;

Field	**Description**
Name	User-defined identifier for the table. All tables should be associated with a unique name. By default, the name is a pointer to the table.
Size	A positive integer or integer constant designated by CONST representing the TABLE size in bytes.
<pin map list>	Pin names exclusively designated by INPUT, OUTPUT, or BIDIR. **DH** stands for the pins in a list to be drivers. **SH** stands for the pins in a list to be sensors. Actual logic high/low states of pins base on the binary value in a table. Individual pins in a list are delimited using commas.

Example 1
```
TABLE DATATAB : 0H80000;
{
   DH(D15,D14,D13,D12,D11,D10,D9,D8,D7,D6,D5,D4,D3,D2,D1,D0);
};
```

The size of a TABLE is calculated in the unit of byte. For example, the size of a TABLE allocated to accommodate programming data for a 512KB flash is 524288 bytes (80000 in hexadecimal), and is independent of the number of pins assigned to a TABLE.

Example 2
```
TABLE ADDRTAB : 0HB00000;
{
  DH(A22,A21,A20,A19,A18,A17,A16,A15,A14,A13,A12,A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1);
};
```

The size of a TABLE allocated to accommodate programming addresses for an 8M bytes flash with 22-bit address bus and 16-bit data bus is calculated as follows.

> For any flash address, it occupies 22-bit memory segment in the TABLE. And it takes 4194304 address iterations (2^{22}) completing programming all data locations on the flash. Thus, the memory size allocated to a TABLE is 92274688 bits, that is 11534336 bytes (B00000 in hexadecimal), near 11M bytes.

To access, load, and save such a huge TABLE would degrade the system performance, and is strongly not recommended. For TABLE size reduction, a TABLE can be split into smaller pieces without adding any overheads to the test. For example, the TABLE ADDRTAB can be divided as follows to reduce significantly the TABLE size from 11M bytes to about 1K bytes.

```
TABLE ADDRTAB : 0H100;
{
  DH(A8,A7,A6,A5,A4,A3,A2,A1);
};

TABLE ADDRTAB : 0H100;
{
  DH(A16,A15,A14,A13,A12,A11,A10,A9);
};

TABLE ADDRTAB : 0H100;
{
  DH(A22,A21,A20,A19,A18,A17);
};
```

TABLEPTR

Type

Keyword

Description

Use the TABLEPTR keyword to declare a table pointer to a table designated by TABLE. A table referenced by more than one table pointer is permitted.

Syntax

TABLEPTR name **=** table-name ;

Field	Description
name	User-defined identifier for the table pointer. All table pointers should be associated with a unique name.
table-name	A name designated by TABLE.

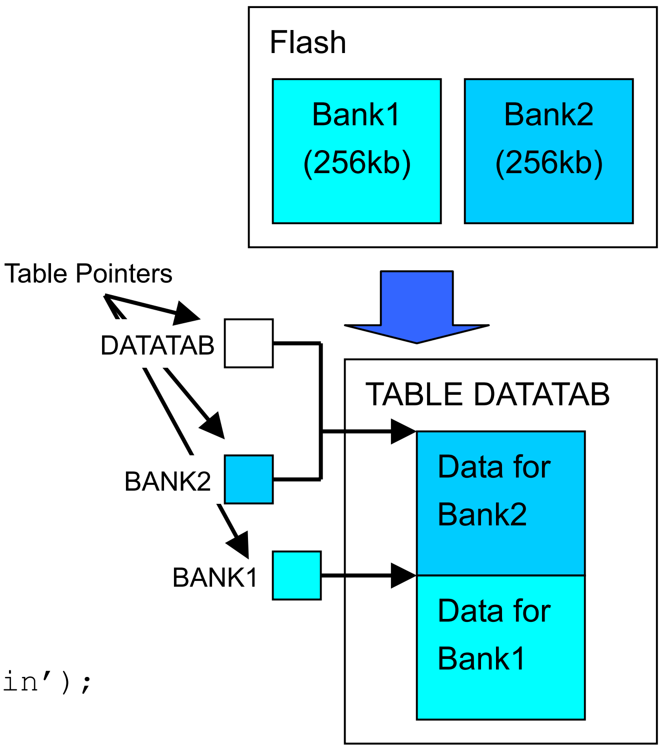
Example

```
TABLE DATATAB : 0H80000;
{
    DH(D15,D14,D13,D12,D11,D10,D9,D8,D7,D6,D5,D4,D3,D2,D1,D0);
};

TABLEPTR BANK1 = DATATAB;
TABLEPTR BANK2 = DATATAB;

BLOCK TEST;
{
    . . . . . . . .
    FL 0H20000
    {
        BANK1+;
        . . . . . . . .
        BANK2+;
        . . . . . . . .
    };
    . . . . . . . .
};
. . . . . . . .

MAIN
    . . . . . . . .
    LOADTABLE(DATATAB,'data.bin');
    USETABLE(BANK2);
    USETABLE(BANK1,0H20000);
    TEST;
    . . . . . . . .
END.
```



USETABLE

Type

Standard routine

Description

The USETABLE routine loads a table pointer designated by TABLE/TABLEPTR to the first test step of a TABLE. This also permits to load a table pointer to the specified test step offset from the first one. Use the USETABLE statement accompanied with table reference statements to execute test steps in a TABLE.

Syntax

USETABLE (table-name [**,** step-offset] **) ;**

Field	Description
table-name	User-defined identifier designated by TABLE/TABLEPTR
step-offset	*Optional* parameter. An integer or integer constant designated by CONST represents the test step offset from the first step (the beginning of a TABLE).

Example

See the example in the TABLEPTR statement.

Non-Test (Utility) Keywords

The keyword listed below defines a user-customized utility subroutine. The implementation for a subroutine is delimited using left and right braces.

SUBROUTINE

The following control-flow command keywords are exclusively used within a SUBROUTINE or the main program to specify the order in which computations are performed.

IF-THEN-ELSE	
FOR-TO-DO	WHILE-DO
GOTO	GOTO-ON

The following two keywords specify the unique execution entry point of a test program and the test plan is bounded. A dot must be appended to END to denote the end of a program.

MAIN-END **.**

Utility Statements

CLOSE

Type
Standard routine

Description
The CLOSE statement closes a stream explicitly established by an OPEN statement. A stream closed cannot be accessed at all in all subsequent I/O statements. Any opened files must be closed before the end of the program.

Syntax
CLOSE (file-descriptor **) ;**

Field	**Description**
file-descriptor	A TEXTFILE or BINARYFILE variable.

Example
See the example in the **OPEN** statement.

FOR-TO-DO

Type
Keyword

Description
The FOR-TO-DO statement iterates a single or compound statement within specified bounds. Refer to **Syntax** below, the FOR loop first evaluates initial-value and assigns it to loop-index. Then it compares loop-index against final-value, and terminates the iteration if loop-index has been over final-value. Thus, loop-body may be executed zero times in the condition of initial-value is initially greater than final-value. The loop-index is increased by one. The nested FOR loop statements are permitted.

Syntax
FOR loop-index **=** initial-value **TO** final-value **DO** loop-body **;**

Field	Description
loop-index	INTEGER/CHAR variable
initial-value	INTEGER/CHAR expression
final-value	INTEGER/CHAR expression
loop-body	Single statement or compound statement

Example
```
FOR I=CURRENTLINE+1 TO MAXLINES DO
   WRITELN(LOGFILE);

FOR I=1 TO EPPIDSIZE DO
{
   WRITE(LOGFILE,STRBUF[2*I-1]);
   WRITE(LOGFILE,STRBUF[2*I]);
   WRITE(LOGFILE,' ');
};
```

GOTO

Type
Keyword

Description
The GOTO statement unconditionally transfers the execution flow of a program to a specified label. A labeled statement must be in the same scope as GOTO statement. That is, GOTO that branches into, out of or between subroutines is not permitted. Also, GOTO that branches into, out of or between compound statements is not permitted. The label must be unique for the entire program.

Syntax
GOTO label ;

Field	Description
label	User-defined identifier on the statement to which a program transfers

Example
```
. . . . . . . .
GOTO ENDOFTEST;
. . . . . . . .
ENDOFTEST:
. . . . . . . .
```

GOTO-ON

Type
Keyword

Description
The GOTO-ON statement transfers the execution flow of a program to a specified label depending on the pass/fail condition of a flag. Refer to **Syntax** below, the condition-value is a Boolean value (true or false) and can be evaluated using standard routines, FAIL or PASS.

Syntax
GOTO label **ON** condition-value **;**

Field	Description
label	User-defined identifier on the statement to which a program transfers
condition-value	A Boolean value evaluated by FAIL or PASS

Example
```
XXXCHECKID;
GOTO YYY ON FAIL(XXXIDFAIL);
. . . . . . . .
GOTO ENDOFTEST;
YYY:
YYYCHECKID;
GOTO ENDOFTEST ON FAIL(YYYIDFAIL);
. . . . . . . .
ENDOFTEST:
. . . . . . . .
```

IF-THEN-ELSE

Type

Keyword

Description

The IF-THEN-ELSE statement, where the ELSE part is optional, is a conditional statement and used to express decisions. Refer to **Syntax** below, the condition-value is evaluated; if it is true (that is, if condition-value has a non-zero value), then-body is executed. If it is false (condition-value is zero) and if there is an ELSE part, else-body is executed instead. It is permitted to concatenate IF-THEN-ELSE statements in series to form a multi-way decision. The condition-values are evaluated in order; if any condition-value is met, the then-body associated with it is executed and this terminates the whole chain. If none of condition-values is satisfied, the else-body is executed if it exists.

Syntax

```
IF condition-value THEN then-body [ ELSE else-body ] ;
IF condition-value THEN
     then-body
ELSE IF condition-value THEN
     then-body
ELSE IF condition-value THEN
     then-body
[ ELSE else-body ] ;
```

Field	**Description**
condition-value	Any Boolean expression
then-body	Single statement or compound statement
else-body	*Optional* statement. Single statement or compound statement

Example

```
// integer value to hexadecimal character conversion
IF VAL>=0 && VAL<=9 THEN
    CH=VAL+’0’
ELSE IF VAL=10 THEN
    CH=’A’
ELSE IF VAL=11 THEN
    CH=’B’
ELSE IF VAL=12 THEN
    CH=’C’
ELSE IF VAL=13 THEN
    CH=’D’
ELSE IF VAL=14 THEN
    CH=’E’
ELSE
    CH=’F’;

IF FAIL(100) THEN
{
    FAILCLR(100);
    WRITELN(’Manufacturer ID Failed’);
};
```

MDLY

Type

Standard routine

Description

The MDLY statement suspends the program execution for the specified time interval in milliseconds. Once the delay interval is expired, the program execution resumes automatically.

Syntax

MDLY (time-interval **) ;**

Field	**Description**
time-interval	A positive integer or integer constant designated by CONST representing the delay interval in milliseconds.

Example

```
MAIN
   . . . . . . . .
   MDLY(1000); // delay 1 sec
   . . . . . . . .
END.
```

MV

Type
Standard routine

Description
The MV statement applies the analog voltmeter to measure the DC voltage on any two points, pin1 and pin2, referring to **Syntax** below. The range of the voltage can be measured for the voltmeter is 0V ~ 100V or –50V ~ 50V. The voltmeter is auto-ranging according to the expected-voltage value specified.

Syntax
MV (expected-voltage **,** pin1 **,** relay1 **,** pin2 **,** relay2 **) ;**

Field	Description
expected-voltage	A floating-point value representing the expected voltage measured.
pin1, pin2	A nail number or pin name designated by `INPUT`, `OUTPUT`, or `BIDIR` representing the pin under measurement.
relay1, relay2	A character literal or constant representing the analog bus used to connect the test nail to the analog voltmeter. The analog buses are: `’A’`, `’B’`, and `’G’`.

Example
```
BLOCK OUT3V;
{
   DH(VID3,VID2) DL(VID1,VID0); // 1100:3V
};

MAIN
   OUT3V;
   MDLY(100);
   VOLT=MV(3.3,O3V,’A’,1,’G’);
   IF VOLT>3.6 || VOLT<3.0 THEN
      WRITELN(’3V OUTPUT FAILED’);
END.
```

OPEN

Type

Standard routine

Description

The OPEN statement establishes a stream over which data can be read from or written to a specified file using READ/READLN or WRITE/WRITELN statements. Refer to **Syntax** below, file-descriptor is a file variable and used to access the file specified by a file-name in all subsequent I/O statements. The file-mode defines I/O operation for a file as read, write, or append. The maximum 10 files are permitted to open simultaneously in TR-8000 series system.

Syntax

OPEN (file-descriptor **,** file-name **,** file-mode **) ;**

Field	Description
file-descriptor	A TEXTFILE or BINARYFILE variable.
file-name	A string literal or constant designed by CONST.
file-mode	A character literal or constant designed by CONST. Allowable modes include as follows. ‘r’ open file for reading only, ‘w’ create file for writing; discard previous contents if any, ‘a’ append; open or create a file for writing at end of file.

Example

```
CONST
   FILENAME='c:\test.log';
   . . . . . . . .

VAR
   STREAM:TEXTFILE;
   . . . . . . . .

MAIN
   OPEN(STREAM,FILENAME,'a');
   . . . . . . . .
   CLOSE(STREAM);
END.
```

STRCAT

Type
Standard routine

Description
This function concatenates a copy of a source string to the end of a target string, and returns the string length after concatenation.

Syntax
[string-length **=**] **STRCAT (** target-string **,** source-string **) ;**

Field	Description
target-string	A character array variable holds the target string to which a source string is appended.
source-string	A string to be concatenated to the end of a target string.
string-length	*Optional*. An integer variable holds the string length of modified target string.

Example
```
VAR
   STRING:CHAR[100];
. . . . . .
STRING[0]=0;
STRCAT(STRING,'THIS IS');
STRCAT(STRING,'A STRING');
```

The content of the variable `STRING` at this point is `THIS IS A STRING`.

STRCHR

Type

Standard routine

Description

This function locates the first occurrence of the specified character in a string, and returns the index of the located character, or 0 if the character does not occur in a string.

Syntax

[index **=**] **STRCHR (** string-to-search **,** character-to-find **) ;**

Field	**Description**
string-to-search	A string that will be searched for the specified character.
character-to-find	A character for which to search in the specified string.
index	*Optional*. An integer variable holds the index of the located character.

Example

```
VAR
   INDEX:INTEGER;
. . . . . .
INDEX=STRCHR('THIS IS A STRING','S');
```

The value of the variable `INDEX` is 4.

```
INDEX=STRCHR('THIS IS A STRING','Y');
```

The value of the variable `INDEX` is 0.

STRLEN

Type
Standard routine

Description
This function computes the length of the specified string. The number of characters, that precede the terminating ASCII NUL byte 0, is returned. The size of a character array is returned if a string stored in an array contains no terminating ASCII NUL byte.

Syntax
[length **=**] **STRLEN (** string **) ;**

Field	Description
string	A string that will have its length computed.
length	*Optional*. An integer variable holds the resulting string length computed by the function.

Example
```
VAR
   STRING:CHAR[10];
   LENGTH:INTEGER;
. . . . . .
STRING='TR-5001 ATE';
LENGTH=STRLEN(STRING);
```
The value of the variable `LENGTH` is 10.

```
STRING[1]=0;
LENGTH=STRLEN(STRING);
```
The value of the variable `LENGTH` is 0.

STRNCPY

Type
Standard routine

Description
This function copies not more than a specified amount of characters from partial source string to a target string, and returns the number of characters actually copied to a target string.

Syntax
[number-copied **=**] **STRNCPY (** source-string **,** copy-from **,** copy-number **,**
target-string **) ;**

Field	Description
source-string	A string from which a specified amount of characters will be copied into a target string.
copy-from	An integer number representing the beginning index of a sub-string in a source string. For example, the string, `TR-5001 ATE`, the sub-string begins at index 9 is `ATE`.
copy-number	An integer number representing the maximum number of characters that will be copied into a target string.
target-string	A character array variable holds a string to which the specified amount of characters from the source sub-string will be copied.
number-copied	*Optional*. An integer variable holds the number of characters actually copied to a target string.

Example
```
VAR
   TARGET:CHAR[100];
. . . . . .
STRNCPY('TR-5001 ATE',1,7,TARGET);
```
The string `TARGET` is `TR-5001`.

STRRCHR

Type

Standard routine

Description

This function locates the last occurrence of the specified character in a string, and returns the index of the located character, or 0 if the character does not occur in a string.

Syntax

[index **=**] **STRRCHR (** string-to-search **,** character-to-find **) ;**

Field	**Description**
string-to-search	A string that will be searched for the specified character.
character-to-find	A character for which to search in the specified string.
index	*Optional*. An integer variable holds the index of the located character.

Example

```
VAR
   INDEX:INTEGER;
. . . . . .
INDEX=STRRCHR('THIS IS A STRING','S');
```

The value of the variable `INDEX` is 11.

STRSCAN

Type
Standard routine

Description
This function converts input from the specified source string into a series of values under the control of a formatting string. The format string contains format specifications that indicate how to convert the input. The function returns the number of input items successfully converted.

Syntax
[number-of-items **=**] **STRSCAN (** source-string **,** format **,** <target-list> **) ;**

Field	**Description**
source-string	A string from which the input to be converted is obtained.
format	A string representing the format that specifies the admissible input sequences and how they are to be converted for assignment. Each conversion specification begins with a % and ends with a conversion character. Valid conversion characters are as follows. D integer. The integer may be in binary (leading 0b), decimal, or hexadecimal (leading 0h). For example, 123, 0h123, 0b0101. C character. F floating-point number possibly containing a decimal point, and an optional exponent field containing an E or e followed by a possibly signed integer. For example, 1.23, 1.3e2. S string of non-white space characters.
target-list	A list of variables, separated by commas, will receive the converted input from a source string.
number-of-items	*Optional*. An integer variable holds the number of input items successfully converted.

Example
```
VAR
   IVAL:INTEGER;
   FVAL:FLOAT;
   STRING:CHAR[100];
. . . . . .
STRSCAN('1234 1.3e2 TR-5001 ATE','%D%F%S',IVAL,FVAL,STRING);
```

The value of the variable `IVAL` is 1234.

The value of the variable `FVAL` is 130.0.

The value of the variable `STRING` is TR-5001.

STRSTR

Type
Standard routine

Description
This function locates the first occurrence of a sequence of characters (sub-string) in a string, and returns the first character index of the located sub-string in a string, or 0 if the sub-string does not occur in a string.

Syntax
[index **=**] **STRSTR (** string-to-search **,** sub-string-to-find **) ;**

Field	**Description**
string-to-search	A string that will be searched to locate the specified sub-string.
sub-string-to-find	A string that will search for in the specified string.
index	*Optional*. An integer variable holds the first character index of the matched sub-string in a string, or 0 if the sub-string is not found.

Example
```
VAR
   INDEX:INTEGER;
. . . . . .
INDEX=STRSTR('TR-5001 IS ATE','8001');
```
The value of the variable `INDEX` is 4.

SUBROUTINE

Type

Keyword

Description

The SUBROUTINE statement defines a user-customized utility routine, which is composed of a sequence of utility statements delimited by left and right braces. Variable declarations and constant definitions are permitted and exclusively accessed within a SUBROUTINE. Subroutines are called only in the main program.

Syntax

SUBROUTINE name [**(** <valid arguments> **)**] **;**
[**CONST** <constant definitions>]
[**VAR** <variable declarations>]
{
        <valid utility statements>
} **;**

Field	**Description**
name	User-defined identifier for a SUBROUTINE. All subroutines must be associated with a unique name.
<valid arguments>	A list of declared variables used in a SUBROUTINE can be called by value or by reference. Individual declared variables are delimited using semicolons.
<constant definitions>	A list of defined constants accessed only in a SUBROUTINE.
<variable declarations>	A list of declared variables accessed only in a SUBROUTINE.
<valid utility statements>	*Optional* statement. Single statement or compound statement.

Example

```
SUBROUTINE TOHEX(VAL:INTEGER; VAR CH:CHAR);
{
    IF VAL>=0 && VAL<=9 THEN
        CH=VAL+'0'
    ELSE IF VAL=10 THEN
        CH='A'
    ELSE IF VAL=11 THEN
        CH='B'
    ELSE IF VAL=12 THEN
        CH='C'
    ELSE IF VAL=13 THEN
        CH='D'
    ELSE IF VAL=14 THEN
        CH='E'
    ELSE
        CH='F';
};

SUBROUTINE LOGRESULT;
VAR
    BYTEBUF:BYTE[20];
    STRBUF:CSTRING[40];
    I,J,HINIBBLE,LONIBBLE:INTEGER;
    LOGFILE:TEXTFILE;
{
    SAVETABLE(DATATAB,BYTEBUF);
    J=1;
    FOR I=1 TO 20 DO
    {
        HINIBBLE=BYTEBUF[I]>>4;
        LONIBBLE=BYTEBUF[I]&0HF;
        TOHEX(HINIBBLE,STRBUF[J]);
        J=J+1;
        TOHEX(LONIBBLE,STRBUF[J]);
        J=J+1;
    };

    OPEN(LOGFILE,'result.log','a');
    FOR I=1 TO 20 DO
        WRITE(LOGFILE,STRBUF[2*I-1],STRBUF[2*I],' ');
    WRITELN(LOGFILE);
    CLOSE(LOGFILE);
};
```

UDLY

Type
Standard routine

Description
The UDLY statement suspends the program execution for the specified time interval in microsecond. Once the delay interval is expired, the program execution resumes automatically. The maximum delay time in microsecond is 65536.

Syntax
UDLY (time-interval **) ;**

Field	Description
time-interval	A positive integer or integer constant designated by CONST representing the delay interval in microsecond.

Example
```
MAIN
    . . . . . . . .
    KDON(PIN1,PIN3,PIN5,PIN7);
    UDLY(500); // delay 500us
    . . . . . . . .
END.
```

WHILE-DO

Type

Keyword

Description

The WHILE-DO statement iterates a single or compound statement as long as the condition-value is true, referring to **Syntax** below. The condition-value is evaluated each time at the start of this iteration, and if satisfied, the loop-body is executed, otherwise, terminates the WHILE statement. Like FOR loop statement, the loop-body may be executed zero times if the condition-value is initially evaluated as a false condition. The nested WHILE statements are permitted.

Syntax

WHILE condition-value **DO** loop-body **;**

Field	Description
condition-value	Any Boolean expression
loop-body	Single statement or compound statement

Example

```
I=1;
WHILE I<20 DO
{
    . . . . . . . .
    I=I+1;
};
```

WRITE / WRITELN

Type
Standard routine

Description
The WRITE/WRITELN statement writes a string (stream of ASCII characters) to the standard output (monitor) or a device specified by a device-descriptor, referring to **Syntax** below. Allowable device includes a line printer designated by LPT or a text file opened by using OPEN statement. The parameter-list is a list of string literals, scalar variables, and string variables, which are delimited by commas. The WRITELN statement automatically appends a new line escape character at end of a string, causing the next WRITE/WRITELN statement to write the data stream in a line next to the current line.

Syntax
WRITE / **WRITELN (** [device-descriptor **,**] <parameter-list> **) ;**

Field	**Description**
device-descriptor	*Optional*. The default device is a monitor. LPT is a reserved identifier representing the line printer connected to the system.
parameter-list	A list of string literals and valid variables to be written to the specific device. The valid data type of variables are scalar types and CSTRING.

Example

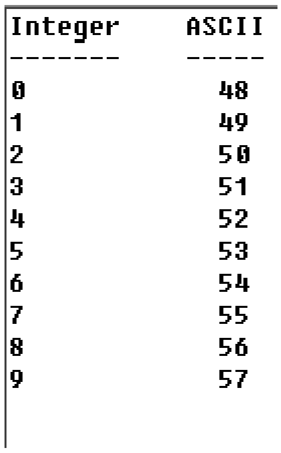
```
CONST
   FILENAME='test.out';

SUBROUTINE TOASCII(INTVAL:INTEGER;VAR ASCIIVAL:INTEGER);
{
   ASCIIVAL=INTVAL+'0';
};

SUBROUTINE TEST;
VAR
   ASCIIVAL:INTEGER;
   STREAM:TEXTFILE;
   I,J:INTEGER;
{
   OPEN(STREAM,FILENAME,'w');
   WRITELN(STREAM,'Integer   ASCII');
   WRITELN(STREAM,'-------   -----');
   FOR I=0 TO 9 DO
   {
      TOASCII(I,ASCIIVAL);
      WRITE(STREAM,I);
      FOR J=1 TO 12 DO
         WRITE(STREAM,' ');
      WRITELN(STREAM,ASCIIVAL);
   }
   CLOSE(STREAM);
};

MAIN
   TEST;
END.
```

The contents of the file "test.out" are as follows at the execution completion of the program.



```
Integer    ASCII
-------    -----
0            48
1            49
2            50
3            51
4            52
5            53
6            54
7            55
8            56
9            57
```

Functional Instrumentation Statements

To provide the flexibility required for external functional instrumentation applications, the test language recognizes two standard instrumentation bus protocols, GPIB and RS-232. GPIB is also known as IEEE-488 Digital Interface for Programmable Instrumentation. RS-232 is COM system in the PC.

GPIB Statements

IBFIND

Type
Standard routine

Description
This function obtains a handle associated with the name of a GPIB interface board (communication between the host PC and a GPIB instrument). A handle is associated with a board in the application program.

Syntax
[status **=**] **IBFIND (NAME =** name **, HANDLE =** handle **) ;**

Field	Description
name	A string representing the name of a GPIB control board. The name prefixes GPIB following a number identifying a board in the system, such as ‘GPIB0’.
handle	An integer variable holds a handle associated with a GPIB control board. The variable is associated with the board specified in the whole test program.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value and 0 represent an error; a positive value represents the status with successful execution.

Example
```
VAR
   STATUS,BD:INTEGER;
MAIN
   STATUS=IBFIND(NAME='GPIB0',HANDLE=BD);
   IF STATUS<=0 THEN GOTO ERROR;
   . . . . . .
ERROR:
   IF STATUS<=0 THEN
      FLAGTESTFAIL;
END.
```

IBDEV

Type
Standard routine

Description
This function obtains the device handle associated with the GPIB instrument specified. The handle is associated with the instrument in the application program.

Syntax
[status **=**] **IBDEV (BOARD =** board-number **,**
PAD = primary-address **,**
[**SAD =** secondary-address **,**]
[**TIMEOUT =** timeout **,**]
HANDLE = handle **) ;**

Field	**Description**
board-number	An integer number identifying a GPIB control board.
primary-address	An integer number representing the address of a GPIB instrument.
secondary-address	*Optional*. An integer number representing the secondary address of a GPIB instrument.
timeout	*Optional*. The predefined integer constant specifies the time limit of IO operations of a GPIB instrument. The valid constants are as follows. The default is T10S.

TNONE	no limit
T10US	10 microsecond
T30US	30 microsecond
T100US	100 microsecond
T300US	300 microsecond
T1MS	1 millisecond
T3MS	3 millisecond
T10MS	10 millisecond
T30MS	30 millisecond
T100MS	100 millisecond
T300MS	300 millisecond
T1S	1 second
T3S	3 second

	T10S 10 second
	T30S 30 second
	T100S 100 second
	T300S 300 second
	T1000S 1000 second
handle	An integer variable holds a handle associated with a GPIB instrument. The variable is associated with the instrument specified in the whole test program.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value and 0 represent an error; a positive value represents the status with successful execution.

Example

```
VAR
   STATUS,ADR:INTEGER;
MAIN
   STATUS=IBDEV(BOARD=0,PAD=5,TIMEOUT=T1S,HANDLE=ADR);
   IF STATUS<=0 THEN GOTO ERROR;
   . . . . . .
ERROR:
   IF STATUS<=0 THEN
      FLAGTESTFAIL;
END.
```

IBONL

Type

Standard routine

Description

This function specifies whether the specified GPIB control board or instrument is to be enabled (online) or disabled (offline) for operation. Taking an instrument offline can be thought of as disconnecting its GPIB cable from other instruments. Putting a control board or an instrument online causes the default configuration settings of a control board or an instrument to be restored.

Syntax

[status **=**] **IBONL (HANDLE =** handle **, ONLINE =** is-online **) ;**

Field	Description
handle	A GPIB control board or instrument handle that is obtained by IBFIND or IBDEV.
is-online	A Boolean value representing whether a board or an instrument is to be online. 1 is online; 0 is offline.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value and 0 represent an error; a positive value represents the status with successful execution.

Example

```
VAR
   STATUS,ADR:INTEGER;
MAIN
   STATUS=IBDEV(BOARD=0,PAD=5,TIMEOUT=T1S,HANDLE=ADR);
   IF STATUS<=0 THEN GOTO ERROR;
   . . . . . .
   STATUS=IBONL(HANDLE=ADR,ONLINE=0); // Disable the instrument
ERROR:
   IF STATUS<=0 THEN
      FLAGTESTFAIL;
END.
```

IBWRT

Type

Standard routine

Description

This function writes a specified number of bytes to an instrument. An instrument is automatically addressed before writing and un-addressed afterwards.

Syntax

[status **=**] **IBWRT (HANDLE =** handle **,**
DATA = command **,**
NWRITE = number-of-write-bytes
[, **NWRITTEN =** number-of-actual-write-bytes] **) ;**

Field	**Description**
handle	A GPIB instrument handle that is obtained by IBDEV.
command	A string representing the command to be written to the instrument. A string literal or character array is allowable.
number-of-write-bytes	The number of bytes to write to an instrument. Usually, it is the length of a string specified in **DATA** field.
number-of-actual-write-bytes	*Optional*. An integer variable holds the number of bytes actually transferred by the write operation.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value and 0 represent an error; a positive value represents the status with successful execution.

Example

```
. . . . . .
COMMAND='ACA 21'; // Set AC voltage 21Vrms
STATUS=IBWRT(HANDLE=ADR,DATA=COMMAND,NWRITE=STRLEN(COMMAND));
IF STATUS<=0 THEN GOTO ERROR;
. . . . . .
ERROR:
. . . . . .
```

IBRD

Type

Standard routine

Description

This function reads a specified number of bytes from an instrument. An instrument is automatically addressed before reading and un-addressed afterwards.

Syntax

[status **=**] **IBRD (HANDLE =** handle **,**
NTOREAD = number-of-read-bytes **,**
DATA = data-buffer
[**, NREAD =** number-of-actual-read-bytes] **) ;**

Field	Description
handle	A GPIB instrument handle that is obtained by IBDEV.
number-of-read-bytes	The maximum number of bytes to read from an instrument.
data-buffer	A character array holds the data read from an instrument. The size of an array in bytes must be greater than or equal to the number of read bytes specified in **NTOREAD** field.
number-of-actual-read-bytes	*Optional*. An integer variable holds the number of bytes actually transferred by the read operation.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value and 0 represent an error; a positive value represents the status with successful execution.

Example

```
VAR
   BUFFER:CHAR[100];
. . . . . .
STATUS=IBRD(HANDLE=ADR,NTOREAD=100,DATA=BUFFER);
IF STATUS<=0 THEN GOTO ERROR;
. . . . . .
ERROR:
. . . . . .
```

IBSTA

Type
Standard routine

Description
This function converts a status number returned by a GPIB function into a meaningful status or error message.

Syntax
IBSTA (STATUS = number **, MESSAGE =** string-buffer **) ;**

Field	**Description**
number	An integer number representing a status or an error.
string-buffer	A character array variable holds the meaningful message for the status or error number specified.

Example
```
VAR
   MESSAGE,COMMAND:CHAR[100];
   STATUS,ADR:INTEGER;
. . . . . .
COMMAND='ACA 21'; // Set AC voltage 21Vrms
STATUS=IBWRT(HANDLE=ADR,DATA=COMMAND,NWRITE=STRLEN(COMMAND));
IF STATUS<=0 THEN GOTO ERROR;
. . . . . .
ERROR:
IF STATUS<=0 THEN
{
   FLAGTESTFAIL;
   IBSTA(STATUS=STATUS,MESSAGE=MESSAGE);
   WRITELN('GPIB ERROR:',MESSAGE);
};
```

RS-232 Statements

OPENCOM

Type
Standard routine

Description
This function opens a COM port, and sets the port parameters as specified.

Syntax
[status **=**] **OPENCOM (PORT =** port-number
[**, NAME =** device-name]
[**, BAUD =** baud-rate]
[**, PARITY =** parity-mode]
[**, NDATA =** number-of-data-bits]
[**, NSTOP =** number-of-stop-bits] **) ;**

Field	**Description**
port-number	An integer number representing a COM port number to operate on. A device name specified in **NAME** field concatenates this number to truly represent a COM port.
device-name	*Optional*. A string representing a prefix of a COM port name, for example, COM1 for COM port 1 using COMM.DRV. COMM.DRV that comes with MS Windows recognizes “COM1” through “COM4” only. Refer to the documentation for your Expended COM Port Board for device names beyond COM4. Default is ‘COM’.
baud-rate	*Optional*. An integer number representing the baud rate for the port specified. Valid values are 110, 150, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 56000, 57600, 115200, 128000, 256000. Default is 9600 baud.
parity-mode	*Optional*. A predefined integer constant representing the parity mode for the port specified. Valid constants are NONE, ODD, EVEN, MARK, SPACE. Default is NONE parity.
number-of-	*Optional*. An integer number representing the number of

data-bits	data bits for the port specified. Valid values are 5, 6, 7, 8. Default is 7 data bits.
number-of-stop-bits	*Optional*. An integer number representing the number of stop bits for the port specified. Valid values are 1 or 2. Default is 1 stop bit.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value represents an error; a positive value represents the status with successful execution.

Example

```
CONST
   PORTNO=1;
VAR
   STATUS:INTEGER;
. . . . . .
// Configure COM1 as 9600 baud, none parity, 8 data bits, 2 stop
// bits
STATUS=OPENCOM(PORT=PORTNO,NDATA=8,NSTOP=2);
GOTO ERROR ON STATUS<0;
. . . . . .
ERROR:
CLOSECOM(PORT=PORTNO);
```

CLOSECOM

Type

Standard routine

Description

This function closes a COM port.

Syntax

[status **=**] **CLOSECOM (PORT =** port-number **) ;**

Field	**Description**
port-number	An integer number representing a COM port number to operate on.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value represents an error; a positive value represents the status with successful execution.

Example

Refer to the example of OPENCOM.

COMRD

Type

Standard routine

Description

This function reads from input queue until the desired number of bytes has been read, a timeout occurs, or termination byte is met, and returns an integer value indicating the number of bytes actually read from queue.

Syntax

[status **=**] **COMRD (PORT =** port-number **,**
NTOREAD = number-of-read-bytes **,**
[**TERM =** termination-byte **,**]
DATA = data-buffer **) ;**

Field	**Description**
port-number	An integer number representing a COM port number to operate on.
number-of-read-bytes	An integer number representing the number of bytes to read from the selected port.
termination-byte	*Optional*. A byte value used to terminate a read operation. When termination byte is compared against a transferred byte to determine the termination of a transfer, only the valid bits (specified in **NDATA** field of **OPENCOM**) are used. For example, for a 7-bit transfer, valid bits are 1 to 7.
data-buffer	A character array variable holds the data read from the selected port.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value represents an error; a positive value represents the number of bytes actually read from the input queue.

Example

```
STATUS=COMRD(PORT=2,NTOREAD=100,DATA=BUFFER);
GOTO ERROR ON STATUS<0;
WRITELN('ACTUALLY READ=',STATUS,'BYTES');
ERROR:
```

COMWRT

Type

Standard routine

Description

This function writes the specified number of bytes to the output queue of the selected port, and returns an integer value indicating the number of bytes actually placed in queue.

Syntax

[status **=**] **COMWRT (PORT =** port-number **,**
DATA = data **,**
NWRITE = number-of-write-bytes **) ;**

Field	**Description**
port-number	An integer number representing a COM port number to operate on.
data	A string representing the data to be written to the selected port.
number-of-write-bytes	An integer number representing the number of bytes to write to the selected port.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value represents an error; a positive value represents the number of bytes actually written to output queue.

Example

```
STATUS=COMWRT(PORT=2,DATA='^~F',NWRITE=3);
GOTO ERROR ON STATUS<0;
STATUS=COMWRT(PORT=2,DATA='^~H',NWRITE=3);
GOTO ERROR ON STATUS<0;
. . . . . .
ERROR:
```

SETCOM

Type

Standard routine

Description

This function is used to control various parameters of the selected COM port.

Syntax

[status **=**] **SETCOM (PORT =** port-number
[**, TIMEOUT =** timeout]
[**, XMODE =** XON-XOFF-mode]
[**, HWHANDSHAKE =** CTS-mode] **) ;**

Field	**Description**
port-number	An integer number representing a COM port number to operate on.
timeout	*Optional*. A floating-point number representing a time limit in second for input/output operations.
XON-XOFF-mode	*Optional*. A predefined integer constant used to enable or disable software handshaking by enabling or disabling XON/XOFF sensitivity on transmission and reception of data. Valid constants are XON or XOFF.
CTS-mode	*Optional*. A predefined integer constant used to enable or disable hardware handshaking mode. Hardware handshaking is used to control the flow of data between the sender and receiver so that the receiver's input queue does not overflow. Valid constants are as follows. OFF — Hardware handshaking is disabled. The CTS line is ignored. The RTS and DTR lines are raised the entire time the port is open. CTS_RTS_DTR — Hardware handshaking is enabled. The CTS line is monitored. Both the RTS and DTR lines are used for handshaking.

	CTS_RTS Hardware handshaking is enabled. The CTS line is monitored. The RTS is used for handshaking. The DTR line is raised the entire time the port is open.
status	*Optional*. An integer variable holds the status returned from the function call. A minus value represents an error; a positive value represents the status with successful execution.

Example

```
CONST
   PORTNO=1;
VAR
   STATUS:INTEGER;
. . . . . .
STATUS=OPENCOM(PORT=PORTNO,NDATA=8,NSTOP=2);
GOTO ERROR ON STATUS<0;
// Set timeout 5 seconds, enable software handshaking
STATUS=SETCOM(PORT=PORTNO,TIMEOUT=5.0,XMODE=XON);
GOTO ERROR ON STATUS<0;
. . . . . .
ERROR:
CLOSECOM(PORT=PORTNO);
```

COMSTA

Type
Standard routine

Description
This function converts a status (an error) number returned by an RS-232 function into a meaningful error message.

Syntax
COMSTA (STATUS = number **, MESSAGE =** string-buffer **) ;**

Field	Description
number	An integer number representing an error.
string-buffer	A character array variable holds the meaningful message for the error number specified.

Example
```
CONST
   PORTNO=1;
VAR
   STATUS:INTEGER;
   MESSAGE:CHAR[100];
. . . . . .
STATUS=OPENCOM(PORT=PORTNO,NDATA=8,NSTOP=2);
GOTO ERROR ON STATUS<0;
STATUS=COMWRT(PORT=PORTNO,DATA='^~F',NWRITE=3);
GOTO ERROR ON STATUS<0;
. . . . . .
ERROR:
CLOSECOM(PORT=PORTNO);
IF STATUS<0 THEN
{
   FLAGTESTFAIL;
   COMSTA(STATUS=STATUS,MESSAGE=MESSAGE);
   WRITELN('RS232 ERROR:',MESSAGE);
};
```