CRISP®/32 ***Language*** *Reference Manual*

CRISP *Software Products*

SQUARE D
GROUPE SCHNEIDER

CRISP®/32
Language Reference Manual
Document number: 500 042 - 004, Rev 3

Document History

Revision	Date	Pages affected / Description of change
1	6/25/91	Initial Release. ECN # 3935
2	9/10/91	Reformat Entire Document. ECN # 3965
3	6/16/95	Document update per ECN#4481

Software Version: **CRISP/32 Rev 3.0 and later**

Table of Contents

Notes:

Introduction

General

This document provides a complete description of the elements of the CRISP/32 language.

This manual is broken down into the following sections.

Section	Description
CRISP Command *(page 3)*	Defines the compiler command and required qualifiers.
Statement Format *(page 7)*	Explains conventions used throughout the manual and defines the special characters and keywords used in CRISP/32 source code.
Declaration Statement *(page 17)*	Defines the keywords used in the CRISP/32 source code to define database variables.
Executable Statements *(page 31)*	Defines the keywords used in CRISP/32 to alter sequential execution of CRISP logic.
Boolean Logic *(page 43)*	Defines the logical entities and operators used in boolean expressions in CRISP/32.
Built-In Functions *(page 47)*	Defines the functions used to track and count event occurrences in CRISP/32.
Appendix A *(page A-1)*	Defines the format of the error messages generated by the compiler, and lists the error messages, defining type, explanation, and user action
Appendix B *(page B-1)*	Defines the ASCII character set.
Appendix C *(page C-1)*	Defines the CRISP reserved logicals.
Appendix D *(page D-1)*	Defines the CRISP valid characters and reserved words.

⚠ WARNING

UNINTENTIONAL EQUIPMENT OPERATION

The application of this software requires expertise in the construction and use of man-machine interfaces and supervisory control systems. Only persons with such expertise should be allowed to install, configure, and apply this product.

Failure to observe this precaution can result in equipment damage, severe personal injury or death.

CRISP Command

Introduction

The CRISP command compiles a CRISP/32 source code file to produce any or all of the following:

- A CRISP/32 source file translated from a CRISP/16 source file
- A database object file
- A MACRO/32 source file representing the CRISP logic code
- A source code listing and error report
- A table of variables with addresses
- A FORTRAN COMMON definition file
- A database link command file.

The CRISP compiler is typically executed via the LGBUILD command. Refer to the CRISP/32 Utilities Manual for more details on the LGBUILD command.

Format

This command has the following format.

CRISP [/qual…] sourcefilespec

sourcefilespec	A specific CRISP source code file. The default file type of .C32 need not be specified. Optional qualifiers (/qual) control the operation of the command and are explained in the following.

Qualifiers

The following qualifiers control the function of the CRISP command. Note that the default qualifiers are identified in the description. All other qualifiers must be explicitly specified.

Qualifier	**Function**
/COMMAND_FILE[=linkfile]	Creates the linkable command file, 'linkfile', that permits all variables in the database image file to be declared as universal symbols. This function is not required under normal circumstances.
/NOCOMMAND_FILE + *This is the default condition.*	Suppresses the creation of the link command file.
/DBOBJ[=objectfile] + *This is the default condition.*	Creates a database object file and writes it to the specified file, 'objectfile'. If no 'objectfile' is specified, the file 'sourcefile.DBO' is generated.
/NODBOBJ	Suppresses the generation of the database object file.

(Continued on next page.)

Qualifiers (cont)

Qualifier	Function
/FORTRAN[=commfile]	Generates a FORTRAN COMMON definition file, 'commfile', that maps the database shareable image. If no 'commfile' is specified, the file 'sourcefile.FOR' is generated.
/NOFORTRAN + *This is the default condition.*	Suppresses the generation of a FORTRAN COMMON definition file.
/LIST[=listfile]	Creates a compile report and source code listing and writes it to the specified file, 'listfile'. If no 'listfile' is specified, the file 'sourcefile.LIS' is generated. Note that LGBUILD will enter the /LIST qualifier (refer to the Utilities Reference Manual for details); however, this can be overridden by explicitly specifying /NOLIST.
/NOLIST + *This is the default condition.*	Suppresses the generation of a list file. Note that LGBUILD will enter the /LIST qualifier (refer to the Utilities Reference Manual for details).
/LOGOBJ[=macrofile] + *This is the default condition.*	Causes the CRISP source code to be compiled into assembler code and written to the specified file 'macrofile'. If no 'macrofile' is specified, the file 'sourcefile.MAR' is generated.
/NOLOGOBJ	Suppresses the creation of the logic MACRO file.
/PREFIX[=["]string["]]	Used in conjunction with the /FORTRAN qualifier to cause the specified characters, 'string' to be appended at the beginning of each variable name in the FORTRAN COMMON file. If quotation marks (") are used, the string is appended exactly as specified. If quotation marks are not used, characters are converted to uppercase. If a string is not specified, the source file name is appended to each variable name.
/NOPREFIX + *This is the default condition.*	Suppress the addition of a prefix string at the beginning of each variable name in the FORTRAN COMMON definition file.

(Continued on next page.)

Qualifiers (cont)

Qualifier	Function
/STATISTICS	Directs the compiler to display statistics about the compile.
/TIME=time	Causes the specified 'time' to be written to the database object file header. Time must be specified in VMS format (i.e., dd-mmm-yyyy hh:mm:ss.cc). If this qualifier is not used, the current date and time is written to the header.
/TRANSLATE=C16	Translates a CRISP/16 logic file 'sourcefile.LOG' into CRISP/32 source code and writes it to a file named 'sourcefile.C32' and then compiles the new source file ('sourcefile.C32') in accordance with the other specified qualifiers and defaults of the CRISP command. Be sure to specify the filename.LOG for CRISP/16 log files. Note that this translation automatically specifies the /TRANSLATE=V20 qualifier.
/TRANSLATE=V20	Translates a CRISP/32 V2.0 thru V2.04 logic source file to V2.7 (and later) syntax and writes it to a new version of the same file. It then compiles the new file in accordance with the other specified qualifiers and defaults of the CRISP command.
/VAR_TABLE	Used in conjunction with the /LIST qualifier to causes a table of variables and their addresses to be created and appended to the list file. This is a default when /LIST is invoked.
/NOVAR_TABLE	Suppresses the generation of the variable table.

Error Messages

Compiler errors result in specific error messages and a summary message. For detailed information regarding error messages, refer to Appendix A.

Notes:

Statement Format

Introduction

Throughout this reference manual, statement formats are shown using a type of 'shorthand' to present the command syntax. The following rules define the conventions used in the format statements.

Keyword required	Uppercase letters and words indicate a keyword that must be entered exactly as shown.
Substitute a value	Lowercase letters and words indicate that you must substitute a word or value where the lowercase letters appear. These substitutions are always fully explained in the text that accompanies the statement.
Optional	Optional arguments and characters are 𝕠𝕦𝕥𝕝𝕚𝕟𝕖𝕕 or enclosed in ⟦ outlined brackets⟧ .
Choose One	Outlined braces (⦃ ⦄) are used to indicate a choice of required arguments. You must substitute one and only one of the arguments enclosed in the braces and separated by vertical bars (|).
Repeat	Ellipsis (…) indicate that the previous argument may be repeated as necessary. Refer to the accompanying text for elaboration when ellipsis are used.
White space	Spaces and horizontal tabs are ignored by the CRISP/32 compiler except where they separate keywords and other identifiers. They may be included where desired to enhance readability, except they may not be placed within a keyword or identifier. Formfeeds may be used to cause page breaks where desired.

Format

The statement format is documented in this manual as follows.

KEYWORD; argument ⟦:n⟧ ⟦,...⟧

KEYWORD;	The keyword must be typed exactly as shown.
argument	Substitute the appropriate word. Acceptable arguments are documented in this section of the page.
:n	Optional argument is shown. In this case, the colon might be a required part of the optional argument and would be explained fully in this part of the documentation.
,	The comma is optional and the ellipsis indicate that more arguments could follow, each separated by a comma.

Examples

The following examples show the use of format shorthand conventions in statements.

Statement	Valid forms of the statement

`KEYWORD;⟦name⟧ ⟦:age⟧ ⟦,…⟧`	`KEYWORD; elizabeth:4,ray:12`
	`KEYWORD; :4`
	`KEYWORD; elizabeth, ray:12`
	`KEYWORD; elizabeth:4`
	`KEYWORD; ,`
	`KEYWORD; ,ray,liza:3,,:5,,,jake`

`KEYWORD;{name|color|tree}`	`KEYWORD; betty`
	`KEYWORD; red`
	`KEYWORD; oak`

Keywords

Certain special characters and keywords are used in CRISP source code as instructions to the CRISP compiler and do not affect the execution of the resultant CRISP logic. The keywords described in this section are as follows.

Keyword	**Description**
! *(page 10)*	Comment delimiter.
\ *(page 11)*	Line continuation symbol.
TITLE; *(page 12)*	Specifies a title for subsequent listing pages.
IDENT; *(page 15)*	Identify the CRISP program version.

Comment Delimiter

The exclamation mark (!) instructs the compiler to ignore all text that follows until it encounters another exclamation mark or a carriage return. This allows you to put comments in the source code that do not affect the executable image.

Example

The following examples show the use of the exclamation mark (!) in CRISP source code.

```
LOGICAL;                 BIT1:TRUE,            ! DESCRIPTION                  !\
                         BIT2,                 ! DESCRIPTION                  !\
                         BIT3:FALSE            ! DESCRIPTION
!
!
!  YOU CAN USE COMMENTS ANYWHERE WITHIN YOUR SOURCE CODE                      !
!  DELIMITED EITHER BY TWO EXCLAMATION MARKS AS THIS LINE IS                  !
!  OR DELIMITED BY AN EXCLAMATION MARK AND A CARRIAGE RETURN, LIKE THIS LINE.
!
FLOAT;                   FLT1: 1.23,\          ! DESCRIPTION
                         FLT2: 2.345,\         ! DESCRIPTION
                         FLT: 5.0              ! DESCRIPTION
```

Line Continuation

The backslash (\) is the line continuation symbol. When the compiler encounters a backslash, it ignores the subsequent carriage return and continues to evaluate the CRISP logic that follows as though the carriage return had not been there.

If the compiler encounters a comment symbol (!) immediately after the backslash, it ignores all text between the exclamation point and the next carriage return that follows.

Example

The following examples show the use of the backslash (\) in CRISP source code.

```
LOGICAL;                  BIT1:TRUE,           ! DESCRIPTION                 !\
                          BIT2,                ! DESCRIPTION                 !\
                          BIT3:FALSE           ! DESCRIPTION


PICB0873 = PIIB873_MAN | PII_TRIM_IDLE | UF_SUSIDLE | PII_TRIM_MOD1 | PII_TRIM_MOD2 |\
                          UF_SUSM1 | UF_SUSM2 | SDIB0867 & ~PII_TRIM_MOD3 & ~UF_SUSM3
```

TITLE;

The TITLE; statement causes a form feed (new page) character to be inserted in the source code listing (sourcefile.LIS), before printing the specified text. Each page is titled with the text string until another title statement is encountered.

Format

The format of the TITLE; statement and a description of each argument is as follows.

TITLE; "text"

"text"	The text string **"text"**, along with the new page number, and time/date stamp are printed at the top of the new page (following a form feed).

Example

The following examples show how the TITLE; statement in a .C32 file causes the title to appear in the .LIS listing.

TITLE;

Example (cont)

```
TITLE; "CRISP/32 Compiler Arrays Test Logic"
IDENT; "V2.7-2"
!
! ARRAYS.C32 -- CRISP/32 compiler test logic for arrays
! Last modification:  19-Sep-1990 10:24
!
LOGICAL; VIRGIN:TRUE, NEW_CLE, ACTIVE, CACDSA, , , , , , , , , PAUSE
!
LOGICAL; DONE, RESET, HITS, PB_ON, DONE_NOT
LOGICAL; BIT1, BIT2, BIT3, BIT4, BIT5, BIT6, BIT7, BIT8
LOGICAL; BITS(4), BITT:TRUE

NUMERIC; NUM1, NUM2, NUM3, NUM4, NUM5, NUM6, NUM7, NUM8
NUMERIC; NUMS(4), NUMT:99

LONG;           LNG1, LNG2, LNG3, LNG4, LNG5, LNG6, LNG7, LNG8
LONG;           LNGS(4), LNGT:40000

FLOAT;          FLT1, FLT2, FLT3, FLT4, FLT5, FLT6, FLT7, FLT8
FLOAT;          FLTS(4), FLTT:0.25, FLT_SPC(30)

TIMER;          TIMER1:10

COUNTER; COUNTER1:3

STRING;         STR1[8]:"String 1"
STRING;         STR2[8]
STRING;         STRS(4)[10], STRT[100]:"That's all, folks!"

LOGICAL; AAAA, TEST(8)

TABLES;

RESTART;


TEST(NUM1) = TEST(NUM2)
TEST(NUM3) = TEST(NUM4-320)
TEST(NUM5) = TEST(NUMT+LNGT)

LABEL; LBL1
```

.C32 File Example

(Continued on next page.)

TITLE;

Example (cont)

```
 CRISP/32 Compiler Arrays Test Logic                       19-SEP-1990 11:33:42  CRISP/32  V2.7-015            Page 1
 V2.7-2                                                    19-SEP-1990 10:24:43  DISK$USER:[NEALE.DEV.COM]ARRAYS.C32;7

    1  TITLE; "CRISP/32 Compiler Arrays Test Logic"
    2  IDENT; "V2.7-2"
    3  !
    4  ! ARRAYS.C32 -- CRISP/32 compiler test logic for arrays
    5  ! Last modification:  19-Sep-1990 10:24
    6  !
    7  LOGICAL; VIRGIN:TRUE, NEW_CLE, ACTIVE, CACDSA, , , , , , , , , PAUSE
    8  !
    9  LOGICAL; DONE, RESET, HITS, PB_ON, DONE_NOT
   10  LOGICAL; BIT1, BIT2, BIT3, BIT4, BIT5, BIT6, BIT7, BIT8
   11  LOGICAL; BITS(4), BITT:TRUE
   12
   13  NUMERIC; NUM1, NUM2, NUM3, NUM4, NUM5, NUM6, NUM7, NUM8
   14  NUMERIC; NUMS(4), NUMT:99
   15
   16  LONG;            LNG1, LNG2, LNG3, LNG4, LNG5, LNG6, LNG7, LNG8
   17  LONG;            LNGS(4), LNGT:40000
   18
   19  FLOAT;           FLT1, FLT2, FLT3, FLT4, FLT5, FLT6, FLT7, FLT8
   20  FLOAT;           FLTS(4), FLTT:0.25, FLT_SPC(30)
   21
   22  TIMER;           TIMER1:10
   23
   24  COUNTER; COUNTER1:3
   25
   26  STRING;          STR1[8]:"String 1"
   27  STRING;          STR2[8]
   28  STRING;          STRS(4)[10], STRT[100]:"That's all, folks!"
   29
   30  LOGICAL; AAAA, TEST(8)
   31
   32  TABLES;
   33

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


 CRISP/32 Compiler Arrays Test Logic                       19-SEP-1990 11:33:42  CRISP/32  V2.7-015            Page 2
 V2.7-2                                                    19-SEP-1990 10:24:43  DISK$USER:[NEALE.DEV.COM]ARRAYS.C32;7

   34  RESTART;
   35
   36
   37  TEST(NUM1) = TEST(NUM2)
   38  TEST(NUM3) = TEST(NUM4-320)
   39  TEST(NUM5) = TEST(NUMT+LNGT)
   40
   41  LABEL; LBL1
   42
```

.LIS Listing Example

IDENT;

The IDENT; statement allows the programmer to more readily track versions of a CRISP program. The string will be displayed in the heading of each page of the listing (.LIS) file. It will also be written to the database object (.DBO) and macro (.MAR) files for reference.

Format

The format of the IDENT; statement and a description of each argument is as follows.

IDENT; "string"

"string"	A string up to 15 characters in length that identifies the version of the source file and its corresponding output files.

Example

```
IDENT; "V1.6-33"
```

Notes:

Declaration Statement

Introduction

The CRISP database is built via *declarations* made in the CRISP name.C32 file. Each declaration is composed of a declarative keyword, followed by one or more variable names.

When a declarative keyword is encountered, space is reserved in the CRISP database for the variables whose names follow. This causes an association to be made between the *variable name* and the *value* that is stored in the corresponding location in the database. All declarative keywords must be placed in the application logic *before* the keyword TABLES;. The keyword TABLES; causes the compiler to build a database based on the declarative keywords and variable names encountered up to that point.

The following defines each declaration statement.

Keyword	**Description**
CONSTANT; *(page 19)*	Declares one or more named constants. These constants do not occupy any space in the database.
COUNTER; *(page 20)*	Declares one or more Counter variables. Reserves two words in the database for each Counter declared.
FLOAT; *(page 21)*	Declares one or more Float variables. Reserves one longword in the database for each Float declared.
LOGICAL; *(page 23)*	Declares one or more Logical variables. Reserves one word in the database for each Logical declared.
LONG; *(page 25)*	Declares one or more Long variables. Reserves one 32-bit longword in the database for each Long declared.
NUMERIC; *(page 26)*	Declares one or more Numeric variables. Reserves one 16-bit word in the database for each Numeric declared.
STRING; *(page 27)*	Declares one or more String variables. Reserves appropriate space in the database for each String declared.

(Continued on next page.)

Keyword	Description
TABLES; *(page 29)*	Indicates the end of the variable declarations.
TIMER; *(page 30)*	Declares one or more Timer variables. Reserves three words in the database for each Timer declared.

Constants

A constant is a value that does not change. Constants may be represented by literal values or you may define a named constant using the CONSTANT statement. Literals are merely the actual numbers and letters that comprise the desired value. CRISP has three types of constants: integer, floating-point, and string.

There are two types of integer constant, decimal and hexadecimal. Decimal constants have the form of an optional sign followed by one or more decimal digits (0 - 9). Hexadecimal constants have an optional sign, a prefix of 0X, and one or more of the characters 0 - 9, A - F. The X and A - F may be either uppercase or lowercase.

Floating-point constants begin with an optional sign followed by zero or more decimal digits before the decimal point. Additional decimal digits may be used after the decimal point to supply the fractional part.

String constants consist of a series of ASCII characters enclosed in quotation marks. To enclose a quotation mark within a string, enter two consecutive quotation marks.

Arrays

All CRISP variable types except counters and timers may be declared individually or as arrays. To declare an array, place the desired dimension enclosed in parentheses immediately after the variable name in the declaration statement. The following example declares an array of 10 longwords in the database.

```
LONG; XYZ(10)
```

Valid array dimensions are integer constants between 1 and 32767, inclusive. Named constants may be used to specify the dimension. The elements of an array are guaranteed to be stored contiguously in the database.

Arrays may be used within the logic code by providing an integer subscript, also enclosed in parentheses. This subscript may be a literal value (5), a named constant, a Numeric or Long variable, or an expression (I-5). Long variables provide better performance for subscripts than Numeric variables.

CONSTANT;

The keyword CONSTANT; causes the compiler to recognize each symbol name encountered after the keyword as a named Constant. Named constants do not occupy any space in the CRISP database. They may, however, be used anywhere that a literal constant may be used.

Each named constant being declared must have an initial value and its data type is taken from that value. Values containing a decimal point declare a floating-point constant; otherwise, it is an integer constant. Integer values between -32768 and 32767 (decimal) are considered to be numeric constants; otherwise, it is a long constant.

Named constants also appear in the variable listing in the .LIS file. Instead of an offset value, the word Constant appears in the offset column.

Format

The Constant declaration statement and an explanation of each statement element follows.

CONSTANT; name:n ⟦, ...⟧

name	The name uniquely identifies the named constant.
:n	The value that is assigned to the named constant.
,	Commas are used to separate named constants in a CONSTANT; declaration statement.

Example

The following examples show valid declarations of named constants. In this example, `NUM_C` will be a numeric constant, `LNG_C` will be a longword constant, and `FLT_C` will be a floating-point constant.

```
CONSTANT; NUM_C:4, LNG_C:0xA000, FLT_C:1234.5
```

COUNTER;

The keyword COUNTER; causes the CRISP compiler to recognize each variable name encountered after the keyword as a Counter variable, reserving two 16-bit words in the CRISP database for each. Counter variables may only be declared individually, arrays are not allowed.

Counter variables are used by the Counter In-line Function to count the number of times that an event occurs and to signal when a specified number of occurrences have taken place. Refer to the Boolean Logic section of this manual for complete details on the implementation of the Counter variable.

Two Addressable Words

The Counter variable is actually two 16-bit words. Each word stores an integer value as described in the following.

SET	This word contains the SET value, the number of events that the counter will count. This value is accessed when the variable name is referenced in CRISP logic. The range of values is from 0 to 65535.
COUNTDOWN	This word represents the number of counts remaining before the counter reaches zero. This value is accessed when the variable name, followed by the prime symbol ('), is referenced in CRISP logic.

Format

The Counter declaration statement and an explanation of each statement element follows.

COUNTER; ⟦name⟧ ⟦:n⟧ ⟦,...⟧

name	The name uniquely identifies the variable. When omitted, an unnamed spare is reserved.
:n	An initial SET value of 'n' may be assigned to the variable. If no initial value is declared, an initial value of zero is assigned.
,	Each comma requires that two words be reserved in the database for the following variable (named or unnamed).

Example

The following demonstrates the use of the keyword COUNTER;.

```
!  **  COUNTER DECLARATION **
COUNTER; SCANCOUNT:100

     ...count up logic statement is as follows

COUNTUP = SCANCOUNT - SCANCOUNT'
```

FLOAT;

The keyword FLOAT; causes the compiler to recognize each variable name encountered after the keyword as a Float variable, reserving 4 bytes in the CRISP database for each.

Float (floating-point) variables are used to represent real numbers in the range of $\pm 0.29\text{x}10^{-38}$ to $\pm 1.7\text{x}10^{38}$ with a precision of approximately seven significant digits. Float variables are expressed as a signed number, followed by a decimal point, followed by optional, additional digits. The decimal point is required in all cases. Plus signs and digits following the decimal point are not required.

Float variables can be used to represent the value of input and output field devices or for 'internal' uses such as scale factors, alarm limits, accumulators, constants for arithmetic calculation, etc.

Float variables are permissible in Math statements, as arguments in logic calls, and in relational expressions. The value of a Float variable may be assigned upon declaration, in a Math statement or as the result of a Call or Recall statement.

Format

The following defines the Float declaration format and statement element.

FLOAT; ⟦name⟧ ⟦(dimension)⟧ ⟦:n.⟧ ⟦,...⟧

name	The name uniquely identifies the variable. When omitted, an unnamed float is reserved.
dimension	An array dimension may be specified to declare a group of variables in the database.
:n.	An initial value of 'n' may be assigned to the variable. Initial values for arrays may be specified by preceding each value with a colon. Two colons implies that the corresponding array element should be initialized to zero.

NOTE

Note that a required decimal point (.) is shown in this statement. If no initial value is declared, Floats are assigned an initial value of zero.

,	Each comma requires that a 4-byte F_floating number be reserved in the CRISP database for the variable that follows (named or unnamed).

FLOAT; (cont)

Example

The following example shows a valid declaration of Float variables.

```
FLOAT;  JTYPEC4:-0.0001189722,\
        JTYPEC3:-0.007877791,\
        JTYPEC2:-0.1809425,\
        JTYPEC1:19.74505,\
        JTYPEC0
FLOAT;  TABLE(4):1.0:2.0:3.0:4.0
```

LOGICAL;

The keyword LOGICAL; causes the compiler to recognize each variable name encountered after the keyword as a Logical (boolean) variable. Note that each Logical statement reserves a multiple of 16 words in the CRISP database. If the statement specifies 1 through 16 variables, 16 words are put into the database. If 17 variables are specified, 32 words are generated.

Versions of CRISP before V2.7 used the keyword INTERMEDIATE to declare logical variables. This keyword may still be used; the output from the compiler will be no different

Logical variables are used within the applications program to represent two-state (i.e., logical) conditions and devices. Logicals can be used for any of the following:

- To represent the state of an input or output device
- To represent a state or condition of the process
- To store the result of a logical expression.

Format

The Logical declaration format and an explanation of each statement element follows.

LOGICAL; ⟦name⟧ ⟦(dimension)⟧ ⟦:TRUE⟧ ⟦,...⟧

name	The name uniquely identifies the variable. When omitted, an unnamed spare is reserved.
dimension	An array dimension may be specified to declare a group of variables in the database.
:TRUE	The default initial value for Logical variables is false. The logical value TRUE can be assigned as shown. The initial value is assigned to the variable upon CRISP system start-up (even before any values are calculated by the applications program). Initial values for arrays may be specified by preceding each value with a colon. Two colons implies that the corresponding array element should be initialized to false.
,	Each comma requires that space be reserved for the following variable (named or unnamed). Space for Logical variables is reserved one word at a time; however, Logicals are always allocated in blocks of 16 regardless of the number actually declared.

LOGICAL; (cont)

Example

The following example shows valid declarations of Logical variables.

```
! MESSAGE MASK DECLARED BELOW
LOGICAL;\
      CRISP_TT:TRUE, CRISP_LP, CRISP_CL, CRISP_SS,\
      CRISP_LOGFIL, CRISP_ALMFIL, CRISP_OUTPUT,CRISP_ERROR,\
      CRISP_DEV00, CRISP_DEV01, CRISP_DEV02, CRISP_DEV03,\
      CRISP_DEV04, CRISP_DEV05, CRISP_DEV06, CRISP_DEV07,\
      CRISP_DEV08, CRISP_DEV09, CRISP_DEV10, CRISP_DEV11,\
      CRISP_DEV12, CRISP_DEV13, CRISP_DEV14, CRISP_DEV15,\
      CRISP_DEV16, CRISP_DEV17, CRISP_DEV18, CRISP_DEV19,\
      CRISP_DEV20, CRISP_DEV21, CRISP_DEV22, CRISP_DEV23
!  MOTOR 1 PUSH BUTTON
LOGICAL; MTR1_PB
!     ALARMS
LOGICAL; ALM_01, ALM_02, ALM_03, ALM_04, ALM_05
```

LONG;

The keyword LONG; causes the compiler to recognize each variable name encountered after the keyword as a Long variable, reserving a longword in the CRISP database for each.

Long variables are 32-bit integer variables used in the application program to store integer values in the range of approximately ± 2 billion (± 2 x 10^9). Since this is the normal size of a VAX integer, Long variables are more efficient than Numeric variables, especially for array subscripts.

The value of a Long variable is established upon declaration, or as a result of a CRISP logic call or an math statement such as the following.

```
LET;,COUNTDOWN=10
```

Format

The Long declaration statement and an explanation of each statement element follows.

LONG; ⟦name⟧ ⟦(dimension)⟧ ⟦:n⟧ ⟦,…⟧

name	The name uniquely identifies the variable. When omitted, an unnamed spare is reserved.
dimension	An array dimension may be specified to declare a group of variables in the database.
:n	An initial value (n) may be assigned to the variable. If no initial value is declared, an initial value of zero will be assigned. A minus sign (-) indicates a negative number. No sign or a plus sign (+) indicates a positive number. Initial values for arrays may be specified by preceding each value with a colon. Two colons implies that the corresponding array element should be initialized to zero.
,	Each comma requires that an additional longword be reserved in the database for the variable which follows (named or unnamed).

Example

The following examples shows valid declarations of Long variables.

```
LONG; REACT1_CALORIES:40000,  REACT2_CALORIES:47000
LONG;LOW_INDEX:-3400
```

NUMERIC;

The keyword NUMERIC; causes the compiler to recognize each variable name encountered after the keyword as a Numeric variable, reserving a word in the CRISP database for each.

Numeric variables are 16-bit integer variables used in the application program to hold integer values in the range of -32768 through 32767. Typically, Numerics are used for accumulators, event counters, small fixed scale factors, constants, or I/ONYX analog output engineering unit variables.

The value of a Numeric variable is established upon declaration, or as a result of a CRISP call or a math statement such as the following example.

```
LET; , COUNTDOWN=10
```

Format

The Numeric declaration statement and an explanation of each statement element is as follows.

NUMERIC; ⟦name⟧ ⟦(dimension)⟧ ⟦:n⟧ ⟦,…⟧

name	The name uniquely identifies the variable. When omitted, an unnamed spare is reserved.
dimension	An array dimension may be specified to declare a group of variables in the database.
:n	An initial value (n) may be assigned to the variable. If no initial value is declared, an initial value of zero will be assigned. A minus sign (-) indicates a negative number. No sign or a plus sign (+) indicates a positive number. Initial values for arrays may be specified by preceding each value with a colon. Two colons implies that the corresponding array element should be initialized to zero.
,	Each comma requires that an additional word be reserved in the database for the variable which follows (named or unnamed).

Example

The following examples show valid declarations of Numeric variables.

```
NUMERIC; TEMP_DISPLAY, NUMBER_OF_LOADS, JOB_CODE
!
NUMERIC; FLOAT_OFFSET:-3
!
NUMERIC; STUFF(6):99:52::33
```

STRING;

The keyword STRING; causes the compiler to recognize each variable name encountered after the keyword as a String variable, reserving the appropriate space (maximum length of a String variable is 130 bytes) in the CRISP database. The text is stored in ASCII format with one character per byte. String variables store text strings that can be written to printers or terminals.

Format

The String declaration statement and an explanation of each statement element follows.

STRING; name ⟦(dimension)⟧ [n] ⟦:"string"⟧ ⟦,...⟧

name	The name uniquely identifies the variable. Name is required.
dimension	An array dimension may be specified to declare a group of variables in the database.
[n]	The integer **n** specifies a maximum number of characters that the string can store. If no value 'n' is specified, an initial string must be specified. Although the number 'n' is optional, the square brackets (**[ ]**) are required. In a string array, all elements of the array have the same maximum length, but each may have a different current length.
:"string"	Strings may be initialized on declaration by appending an initial string value, enclosed in quotation marks, all preceded by a colon (:). To include a quotation mark within a string, type two consecutive quotation marks.
,	Commas (,) are used to separate String variables and their values, in a declaration.

NOTE

Either the element n or :"string" must be specified when declaring a string. For string arrays, it is recommended that the size be explicitly specified. Otherwise, the maximum size of each string in the array will be set to the length of the first initial value located and any subsequent initial value strings will be truncated to that length if they are longer.

STRING; (cont)

Control Characters

Control characters can be included in text strings. A character preceded by the @ character is converted to an ASCII control character. For example @J is interpreted as Control-J (ASCII 10). The @ symbol works by subtracting 64 from the decimal ASCII value of the character. Refer to the Appendix B for a complete list of ASCII characters.

If the actual '@' character must appear in a text string, @@ must be used. For example, the following CRISP code will be interpreted "TOOL @ CORRECT RPM." followed by a carriage return and a line feed.

```
   STRING;MESSAGE1[]:"TOOL @@ CORRECT RPM.@M@J"
```

Example

The following examples show valid declarations of String variables.

```
STRING; MESSAGE_1[32], TEMPS(8)[20]

STRING; MESSAGE_2[]:"EXCESSIVE TEMPERATURE"

STRING; DOUBLE_STAR[]:"**", OVEN_2[]:"OVEN 2",\
        ALERT_MSG[21], TIME_OUT[20]:"PROCESS OVER"

STRING; STUFF(4)[8]:"First":"Next"::"Last"
```

TABLES;

The keyword TABLES; indicates that no more variable declarations follow. When the CRISP compiler encounters the keyword TABLES;, it builds the CRISP database.

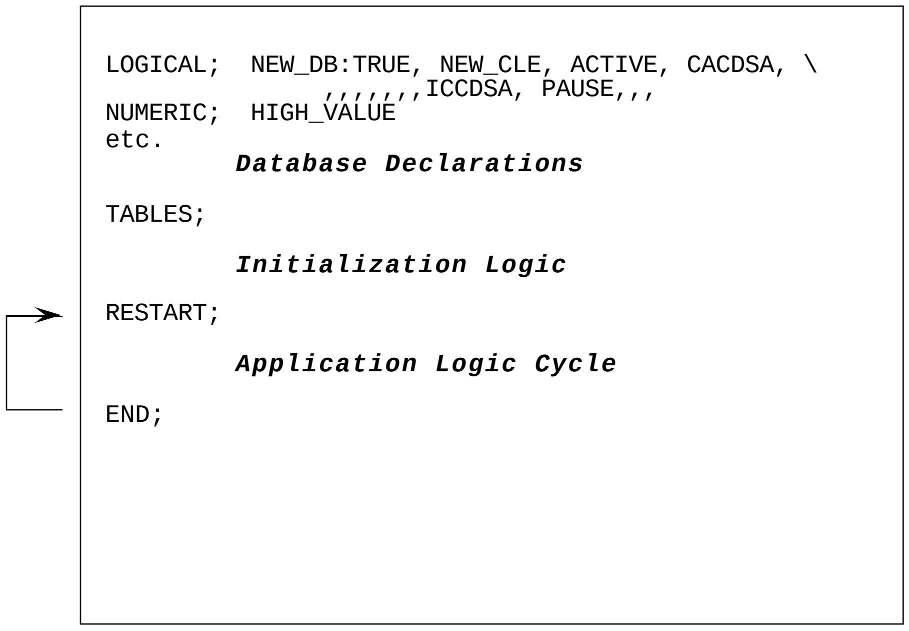
Format

The format of the Tables statement is as follows.

TABLES;

Example

The following example shows the program structure of a typical CRISP/32 logic program.



```
LOGICAL;  NEW_DB:TRUE, NEW_CLE, ACTIVE, CACDSA, \
                 ,,,,,,,ICCDSA, PAUSE,,,
NUMERIC;  HIGH_VALUE
etc.
         Database Declarations

TABLES;

         Initialization Logic

RESTART;

         Application Logic Cycle

END;
```

TIMER;

The keyword TIMER; causes the CRISP compiler to recognize each variable name encountered after the keyword as a Timer variable, reserving three 16-bit words in the CRISP database for each. Timer variables may only be declared individually, arrays are not allowed.

Timer variables are used in boolean Timer functions to measure the elapsed time of an event. Refer to the Boolean Logic section of this manual for complete details.

The CRISP Timer variable is actually three 16-bit words. Each word stores an integer value as described in the following.

Word	**Function**
SET	The number of seconds for which the timer is set. Maximum value is 65,535 seconds (approximately 1092 minutes). It is this value that is accessed when the variable name is referenced in CRISP logic.
COUNTDOWN	The seconds remaining before the timer times-out. This value is accessed when the variable name, followed by the prime symbol ('), is referenced in CRISP logic.
STATUS	Reserved.

Format

The Timer declaration format and an explanation of each statement element follows.

TIMER; ⟦name⟧ ⟦:n⟧ ⟦,…⟧

name	The name uniquely identifies the variable. When omitted, an unnamed spare is reserved.
:n	An initial SET value of (n) may be assigned to the variable. If no initial value is declared, an initial value of zero is assigned.
,	Each comma requires that an additional three words be reserved in the Timer table for the variable that follows (named or unnamed).

Example

The following example shows the use of the keyword TIMER;.

```
! DECLARATION
TIMER; EG_TIMER:180        ! SET FOR 3 MINUTES
```

Executable Statements

Introduction

The CRISP executable statements are used to develop the application-specific code.

Statement	Description
Unconditional Assignment *(page 33)*	Assigns the results of an arithmetic or a logical expression to a variable.
Conditional Assignment *(page 34)*	Conditionally assigns the results of an arithmetic or a logical expression to a variable.
CALL *(page 36)*	Executes a Function call upon change of key from false to true.
END *(page 37)*	Returns to RESTART;.
JUMP *(page 38)*	Jumps to LABEL; when key is True.
LABEL *(page 39)*	Identifies the target for a JUMP statement.
MESSAGE *(page 40)*	Causes a text string and a time and date stamp, to be sent to an output device or file.
RECALL *(page 41)*	Executes a Function call when key is true.
RESTART *(page 42)*	Marks the start of the CRISP logic cycle.

Conditional Expression

Many CRISP statements allow or require a logical or relational expression whose results determine whether the statement should be executed during this pass of the logic program. Some statements are always executed if the expression result is true. These are called 'level-triggered' statements. Others are only executed on transition of the expression from one state to the other (usually from false to true). These are called 'edge-triggered' statements.

(Continued on next page.)

Conditional Expression (cont)

For each edge-triggered statement, CRISP maintains a 'history' of the results of the condition expression from the preceding logic pass. By default, these history values are false on the first pass. However, in some cases, it is desirable for this value to be true for the first logic pass. Adding the keyword TRUE enclosed in brackets ([TRUE]) immediately before the condition expression will accomplish this (refer to the following example).

```
SET;[TRUE]~DONE, X = X + 1
```

Note that the history is kept on the result of the entire conditional expression. Therefore, you must avoid using edge-triggered statements inside a loop where the condition expression includes an array variable whose subscript includes the loop index variable.

⚠ CAUTION

Use of edge-triggered statements in loops can result in an infinite loop. Use extreme caution when using these statements.

Array References

CRISP array variables may be used where a CRISP variable is allowed. When an array variable is used in an executable statement, a subscript must be specified. The subscript may be an integer constant, a Numeric or Long variable, or an expression whose result is a Numeric or Long value. Valid subscripts range from zero to one less than the array dimension (i.e., zero to seven for an array dimension of eight). Note that the compiler will check the validity of a constant subscript, but it does not automatically generate code to perform run-time checking of variable subscripts. The following are some examples of array usage.

```
FLOAT; FLTS(8)

FLTS(0)                    ! Valid
FLTS(7)                    ! Valid
FLTS(-1)                   ! Invalid (negative)
FLTS(8)                    ! Invalid (too large)

FLTS(NUM1)                 ! Variable subscript
FLTS(LNG1+NUM3)            ! Expression subscript
FLTS(NUMS(2)*NUM1)         ! Expression subscript
```

Evaluation of Expressions

Arithmetic and logical expressions are evaluated left to right. When an operator is encountered, that operation is performed between the entities to the right and left of the symbol.

Parentheses are used to indicate operations that must be performed in an order *other than* left to right, as follows.

```
PCNTOP = (TV101 + TV301) * (TEMP101 / CNST3)
```

Parentheses may be nested within parentheses as follows.

```
PCNTOP = (TV101 + (TV301 - OVR) + TV301)
```

Division By Zero

Division by zero is invalid and the results produced by a division by zero in CRISP applications are unpredictable. Therefore, your code should include a check for a zero divisor such as the following.

```
LET; TPT101 & (DIVISR<>0.0), DIVIDND = VRBLE101 / DIVISR
```

Unconditional Assignment Statements

Unconditional assignment statements are used to assign the results of an arithmetic expression or a logical expression to an appropriate variable.

Format

The format of the unconditional assignment statement and a description of each component are as follows.

variable = expressn

variable The 'variable' to the left of the equal sign will contain the result of the expression to the right of the equal sign. It must be of a type that is compatible with the expression (i.e., a Logical for logical expressions).

expressn An arithmetic expression is any combination of Numeric, Long, Float, Timer, or Counter variables and arithmetic operators. The four arithmetic operators are as follows.

+	ADD
-	SUBTRACT
*	MULTIPLY
/	DIVIDE

Logical expressions are described in the Boolean Logic section.

Example

The following are examples of typical assignment statements.

```
OVEN_READY = COOK_PROC  &  TEMP_OK | MANUAL

NOT_READY = ~READY

PCNTOP = (TV101 + TV301) * (TEMP101 / CNST3)
```

Conditional Assignment Statements

The conditional assignment statements are used to assign the results of an arithmetic expression or a logical expression to a specified variable.

Format

The format of the conditional assignment statement and a description of each component are as follows.

LET; cond, variable = expressn
SET; [[TRUE]] cond, variable = expressn

keyword The keyword determines how the condition expression is to be interpreted. Two keywords are used.

Keyword	Resulting Action
SET;	The statement is evaluated only upon the transition of the conditional key from false to true. Note that this transition must hold state for longer than the logic cycle.
LET;	The statement is evaluated once each logic scan as long as the conditional key is true.

cond The condition expression is used to determine if the statement is to be executed. The expression is any valid Logical variable, boolean expression, or relational expression. The Let statement is level-triggered, while the Set statement is edge-triggered.

variable The 'variable' to the left of the equal sign will contain the result of the expression to the right of the equal sign. It must be of a type that is compatible with the expression (i.e., a Logical for logical expressions).

expressn An arithmetic expression is any combination of Numeric, Long, Float, Timer, or Counter variables and arithmetic operators. The four arithmetic operators are as follows.

+	ADD
-	SUBTRACT
*	MULTIPLY
/	DIVIDE

Logical expressions are described in the Boolean Logic section.

CAUTION

The SET Statement should not be used before the RESTART keyword. Use the LET Statement.

CALL;

The keyword CALL; is used to direct the CRISP/32 compiler to a block of external code, a Function call, that performs a specific operation before returning to the execution of the application program.

Format

The format of the CALL Statement is shown below followed by a description of each argument.

CALL; function_call, ⟦[TRUE]⟧ cond, arg ⟦,...⟧

function_call	This must be the name of a CRISP/32 Function call. Function calls are blocks of code which are stored external to the main logic. This enables a single block of code, the Function call, to perform the same function more than once within a single application without requiring that programming statements be repeated each time. Each Function call is explained in detail in the CRISP Function Calls Reference Manual.
cond	The condition expression is used to determine if the statement is to be executed. The expression is any valid Logical variable, boolean expression, or relational expression. This statement is edge-triggered.
arg	A CRISP variable that contains a value to be passed to the Function call or that is to receive a value returned by the call. Each Function call has its own list of arguments. Refer to the CRISP Function Calls Reference Manual for specific details. Note that a maximum of 250 arguments may be passed to any function.
,	The comma is used to separate additional arguments when multiple arguments are required by the Function call.

Argument Checking

When a Function Call is executed, CRISP checks the argument list only if the CACDSA bit is false (refer to the CRISP Reserved Flags in Appendix C of this manual). The compiler does not check the argument list for errors. You should set CACDSA false when debugging your program and true when in production mode (to achieve optimum speed). When debugging a program, set CACDSA false then execute your function call. A message will be printed on the CRISP$TT device if an error is detected.

⚠ CAUTION

The CALL Statement should not be used before the RESTART keyword. Use the RECALL Statement.

END;

The keyword END; indicates the end of the CRISP logic cycle. The Application Program then pauses until the next scheduling interval at which time it resumes execution at the statement immediately following the RESTART; keyword.

⚠ CAUTION

Failure to execute the END; statement prevents the application program from being rescheduled, thus, I/ONYX I/O will not be performed.

Format

The format of the END; statement is as follows.

END;

JUMP;

The keyword JUMP; directs program execution to a specified label where sequential execution of the logic continues.

Format

The format of the logic statement and a description of each argument follows.

JUMP; label, cond

label	This must be a valid label (refer to the LABEL; keyword section of this manual).
cond	The condition expression is used to determine if the statement is to be executed. The expression is any valid Logical variable, boolean expression, or relational expression. This statement is level-triggered.

LABEL;

The keyword LABEL; declares the text string that follows as a CRISP label. When the CRISP compiler encounters the LABEL; keyword it associates the address at which the declaration occurs with the text string provided. This enables the use of the label (text) in a JUMP statement to redirect logic execution to the location of the label.

Format

The format of the Label statement and a description of each argument follows.

LABEL; labelname

labelname A string of characters (up to 30 characters permitted, refer to Appendix D for reserved words and valid characters). Note that a label name may not be the same as a variable name.

Example

Refer to the following example for a possible application of the LABEL keyword, and the corresponding JUMP statement.

```
    .
    .
    .
JUMP; LABEL1, ALARM1
    .
    .
    .
JUMP; LABEL2, ALARM2
LABEL; LABEL1
    .
    .
    .
LABEL; LABEL2
    .
    .
    .
```

⚠ CAUTION

Jumping backwards may place the logic process in a loop. This will prevent the I/ONYX I/O from updating because the END; statement will never be executed.

MESSAGE;

The MESSAGE; statement causes a text string and a time and date stamp to be sent to an output device or file.

Format

The format of the logic statement and a description of each argument is as follows.

MESSAGE; ⟦[TRUE]⟧ cond, "txtstrng"

cond	The condition expression is used to determine if the statement is to be executed. The expression is any valid Logical variable, boolean expression, or relational expression. This statement is edge-triggered.
"txtstrng"	The text string must be explicitly specified and enclosed in quotation marks ("). The string may be up to 130 characters long.

How It Works

The Message statement causes the specified text, preceded by the date and time, to be sent to the print server along with a snapshot of the Message Mask. The print server reads the message mask , then routes the message to the appropriate device or file. Refer to the Function Call SETMSG for complete details on the message mask.

Example

The following shows valid syntax for the Message statement, followed by the printed message.

```
MESSAGE;TL_ALARM,\
"TEMPERATURE HAS EXCEEDED 120 @M@J CHECK BURNER IMMEDIATELY@M@J"
```

```
12:23:01 02-08-87 TEMPERATURE HAS EXCEEDED 120
CHECK BURNER IMMEDIATELY
```

RECALL;

The keyword RECALL; is used to direct the CRISP/32 compiler to a block of external code, a Function call, that performs a specific operation before returning to the execution of the application program.

Format

The format of the RECALL Statement and a description of each argument follows.

RECALL; function_call, cond, arg [,…]

function_call	This must be the name of a CRISP/32 Function Call. Function Calls are blocks of code that are stored external to the main logic. This enables a single block of code, the Function Call, to perform the same function more than once within a single application without requiring that programming statements be repeated each time. Each Function call is explained in detail in the CRISP Function Calls Reference Manual.
cond	The condition expression is used to determine if the statement is to be executed. The expression is any valid Logical variable, boolean expression, or relational expression. This statement is level-triggered.
arg	A CRISP variable that contains a value to be passed to the Function call or that is to receive a value returned by the call. Each Function call has its own list of arguments. Refer to the CRISP/32 Function Calls Reference Manual for specific details. Note that a maximum of 250 arguments may be passed to any function.
,	The comma is used to separate additional arguments when multiple arguments are required by the Function call.

Argument Checking

When a Function Call is executed, CRISP checks the argument list only if the CACDSA bit is false (refer to the CRISP Reserved Flags in the Appendix C of this manual). The compiler does not check the argument list for errors. You should set CACDSA false when debugging your program and true when in production mode (to achieve optimum speed).

RESTART;

The keyword RESTART; indicates where the repetitive logic cycle begins. Each CRISP application program has a scheduling interval specified during logic configuration. At each interval after the first, execution is begun with the first statement after the RESTART; statement.

Format

The format of the RESTART statement is as follows.

RESTART;

Boolean Logic

Introduction

Boolean expressions are composed of *logical entities* and *operators*. Logical entities are defined as anything that can be evaluated as a single logical state, represented by a logical value of true or false. The operators are the symbols that relate the logical entities.

Boolean Operators

The Boolean operators are the symbols that define the relationships in boolean logic. These symbols are as follows.

Symbol	**Description**	**Truth Table**
& (ampersand)	AND Operator - Entity to the left *and* the right of this symbol must be true for the relationship to be true.	$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$
| (vertical bar)	OR Operator - Entity to the left *or* the right must be true for the relationship to be true.	$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}$
^ (caret)	XOR Operator - If either entity but not both is true, the relationship is true. If both are true or both are false, the relationship is false.	$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$
~ (tilde)	NOT Operator - Entity to the right is evaluated, then negated (true becomes false; false becomes true).	

Keywords

The keyword TRUE may be used in a boolean expression when a logical true value is to be expressed. Similarly, the keyword FALSE may be used in a boolean expression when a logical false value is to be expressed.

Logical Variables

The most basic element of boolean logic is the Logical variable. The Logical variable is used to represent the condition of digital (2-state) field devices connected to the I/O system. It is also used internally to represent any 2-state condition existing within the context of the CRISP logic.

(Continued on next page.)

Logical Variables (cont)

Logical variables are declared via the LOGICAL; keyword (refer to Declarative keywords in this manual) and, when appropriate, linked to a specific I/O point via the I/ONYX configurator (refer to the CRISP/32 I/ONYX Configurator Reference Manual).

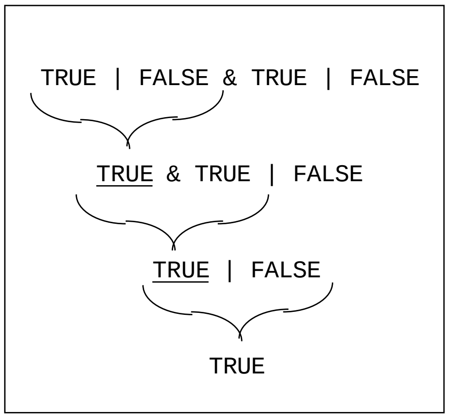
Logical Expressions

A logical expression consists of one or more of the following entities combined by a boolean operator.

- Logical variable
- Relational expression
- Built-in logical function
- Logical or relational expression enclosed in parentheses

A complex logical expression is, *itself*, a logical expression. It is composed of logical entities that can be evaluated as true or false. The expression, as a whole, is reduced to a single logical entity.

CRISP application programming is evaluated from top to bottom, left to right (as it appears in the source code listing). Consequently, boolean (and relational) expressions are evaluated from left to right. When a boolean operator is encountered, the operation is performed between the entity to the left and right of the operator. The following example shows how expressions are evaluated.



Often, it is necessary to indicate boolean operations that must be evaluated in an order *other than* left to right. Parentheses are used to identify boolean (or relational) expressions *within* boolean expressions. For example, refer to the following.

```
OPEN & OFF | (EMPTY | POSITIONED)
```

This use of parentheses ensures that the expression (`EMPTY | POSITIONED`) is evaluated before it is ORed with the result of the operation `OPEN & OFF`.

Multiple levels of parentheses may exist in a boolean expression as shown in the following example.

```
OPEN & OFF | (EMPTY | (DONE & CHARGED) & POSITIONED)
```

Relational Expressions

The relational expression is a logical entity whose value depends on the relationship between the numerical values of two specified variables. The relational expression is true when the values have the relationship indicated by the relational operator.

Format

The format of the relational expression and a description of each of its components are as follows.

arg1 op arg2

arg1, arg2 Arg1 and arg2 are CRISP variables, constants, or arithmetic expressions enclosed in parentheses. The following variable types are permitted: Numeric, Long, Float, Timer, and Counter.

NOTE

When using a constant variable and the other argument is a CRISP float variable, it is recommended that the constant be specified as a float (e.g., ARG1==1.0).

op Relational Operator. The following are valid operators.

Operator	**Description**
==	equal to
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Example

The following examples illustrate relational expressions.

Expression	**Evaluates TRUE when...**
`X >= Y`	`X` is greater than or equal to `Y`
`X < Y`	`X` is less than `Y`
`X == Y`	`X` is equal to `Y`
`X <> Y`	`X` is not equal to `Y`
`X == (Y-5)`	`X` is equal to `(Y-5)`

Notes:

Built-In Functions

Introduction

Built-in functions may be used directly in CRISP expressions where a variable of the same type as the return value of the function may be used. They may have one or more arguments and return a single value -- the function results.

The functions described in this section are as follows.

Function	**Description**
Counter Function *(page 48)*	Used to track the number of times that a specific event has occurred.
Timer Function *(page 50)*	Used to indicate whether the elapsed time of a specific event has occurred.

Counter Function

The Counter function returns a logical value indicating the number of times that a specific event has occurred. The Counter function is evaluated at run time as a single logical value (true when the Counter has counted down to zero and false at all other times).

Format

The format of the Counter function and a description of each of its components are as follows.

COUNTER (reset, event, countername)

reset	Logical variable or logical expression that resets and enables the Counter function. When 'reset' changes from false to true, the Counter is reset. Reset must be true for the counter to count down.
event	Logical variable or boolean expression that triggers the count down. When 'event' changes from false to true, the COUNTDOWN value decrements by one. Note that this transition must hold state for longer than the logic cycle.
countername	Name of the Counter variable.

How It Works

A CRISP Counter variable is actually two 16-bit words. Each word stores an integer value.

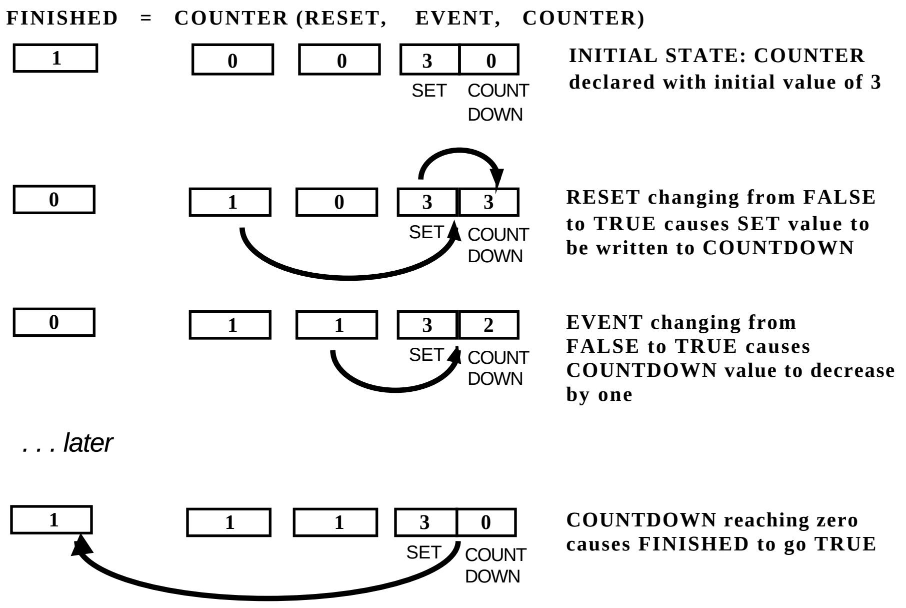
SET	The number of events that the counter will count. Any integer from 1 to 65,535 is valid.
COUNTDOWN	The number of counts remaining before the counter reaches zero.

The SET value must be set to the number of events that the you plan to count. This causes the COUNTDOWN value to be assigned the same value. Initially, the Counter function is evaluated false. Each time 'event' goes from false to true, the COUNTDOWN value is decremented by one. When the COUNTDOWN value reaches zero, the Counter function becomes true. When 'reset' changes from true to false, the SET value is written to the COUNTDOWN value of the Counter variable. The counter function will only count down when the 'reset' value is true.

Counter Function

How It Works (cont)

The following diagram shows the relationships between the elements in the Counter function.


FINISHED = COUNTER (RESET, EVENT, COUNTER)
1
0
0
3
0
SET
COUNT DOWN
INITIAL STATE: COUNTER declared with initial value of 3
0
1
0
3
3
RESET changing from FALSE to TRUE causes SET value to be written to COUNTDOWN
0
1
1
3
2
EVENT changing from FALSE to TRUE causes COUNTDOWN value to decrease by one
. . . later
1
1
1
3
0
COUNTDOWN reaching zero causes FINISHED to go TRUE

Example

A typical counter operation is shown in the following example.

```
! This Counter will reset upon counting down, and will operate
! as long as the ON pushbutton is TRUE.
!Declaration of counter
COUNTER; COUNTER1:3                          ! Declare counter; initialize
                                             ! counter at 3
LOGICAL; PB_ON, DONE, RESET, HITS
                .
                .
                .
! Counter programming
DONE = COUNTER (RESET, HITS, COUNTER1)       ! Count the number of HITS, when
                                             ! HITS = 3, set DONE to TRUE.
RESET = PB_ON & ~DONE                        ! Recycle counter as long as
                                             ! process is ON.
HITS = FALSE                                 ! Resets HITS each cycle.
```

Timer Function

The Timer function returns a logical value indicating whether the elapsed time since a specific event has occurred. The Timer function is evaluated at run time as a single logical value (true when the Timer has timed out and false at all other times). The timer can be interrupted when desired without resetting the elapsed time counter.

Format

The format of the Timer function and a description of each of its components are as follows.

TIMER (reset, enable, timername)

reset	Logical variable or boolean expression that determines when to reset the timer. When 'reset' is false, the system resets the Timer to its assigned SET value.
enable	Logical variable or boolean expression that determines when the Timer can operate. 'Enable' must be true for the Timer to count down.
timername	Name of the Timer variable.

How It Works

A CRISP Timer variable is actually three 16-bit words. Each word stores an integer value.

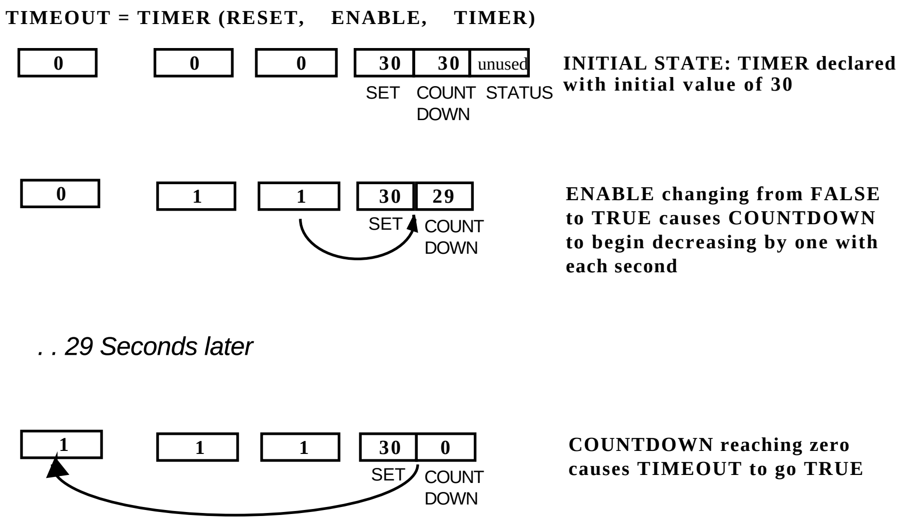
SET	The number of seconds that the Timer is to count. Any integer from 1 to 65,535 is valid.
COUNTDOWN	The number of seconds remaining before the Timer times out.
STATUS	Reserved for future use.

The SET value determines the number of seconds that you plan to count. When 'reset' is false, the SET value is written to COUNTDOWN. When 'reset' is true, the Timer is free to start counting, but will not start until 'enable' goes from false to true. As long as both 'enable' and 'reset' are true, the COUNTDOWN value is decremented by one each second. When the COUNTDOWN value reaches zero, the Timer expression becomes true.

Timer Function

How It Works (cont)

The following diagram shows the relationships between the elements in the Timer function.


TIMEOUT = TIMER (RESET, ENABLE, TIMER)
0
0
0
30
30
unused
SET
COUNT DOWN
STATUS
INITIAL STATE: TIMER declared with initial value of 30
0
1
1
30
29
SET
COUNT DOWN
ENABLE changing from FALSE to TRUE causes COUNTDOWN to begin decreasing by one with each second
. . 29 Seconds later
1
1
1
30
0
SET
COUNT DOWN
COUNTDOWN reaching zero causes TIMEOUT to go TRUE

Example

A typical Timer operation is shown in the following example.

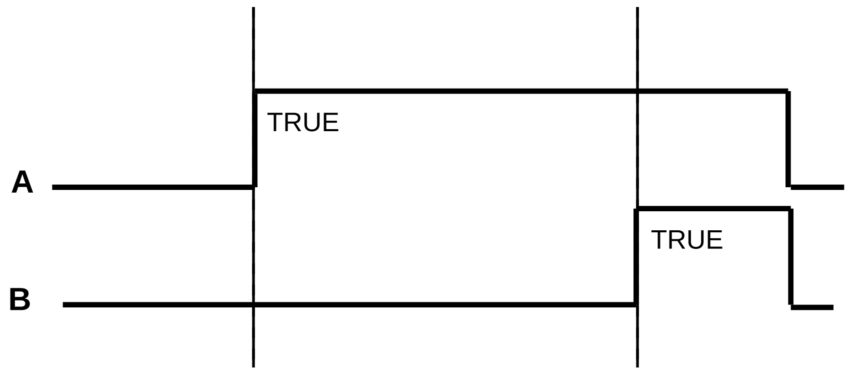
```
! This Timer is programmed to reset upon timing out.
!
! Declaration of TIMER
TIMER; TIMER1:30                            ! Declare timer SET to 30 sec.
LOGICAL; GO:TRUE                            ! Declare GO
                 .
                 .
                 .

! Timer Expression
GO = TIMER (~GO, TRUE, TIMER1)              ! When TIMER1 times out set GO
                                            ! CLEAR GO to start timer.
```

Timer Function

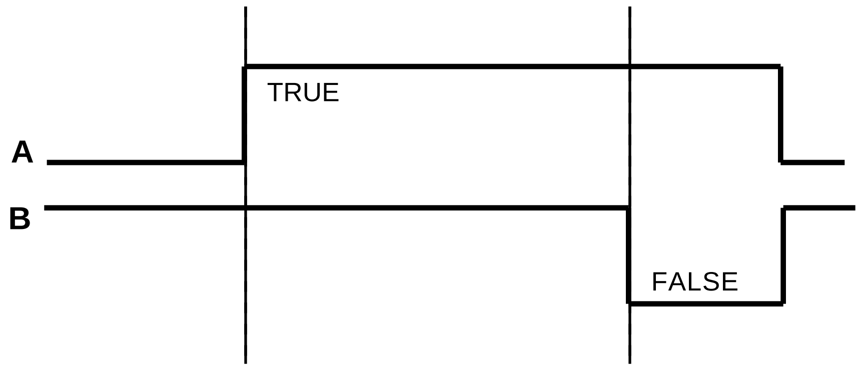
Example (cont)

Other Timer functions are shown schematically in the following examples.

```
FUNCTION:  TIME DELAY "ON TO ON"
STATEMENT:  B = TIMER (A, A, TIMER_01)
```



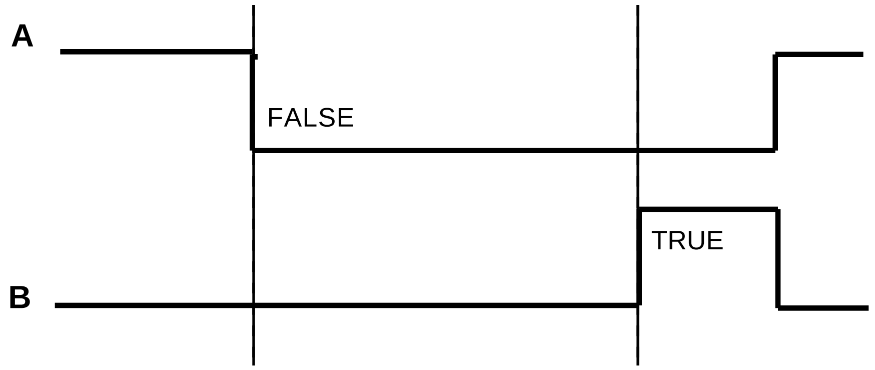
```
FUNCTION:  TIME DELAY "ON TO OFF"
STATEMENT:  B = ~TIMER (A, A, TIMER_02)
```



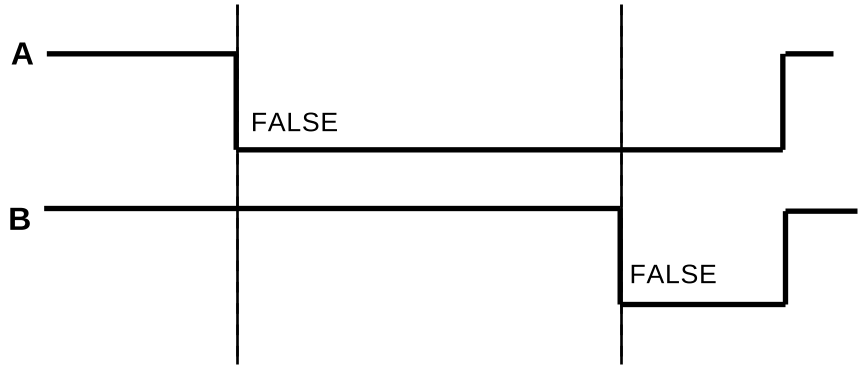
Timer Function

Example (cont)

```
FUNCTION:  TIME DELAY "OFF TO ON"
STATEMENT:  B = TIMER (~A, ~A, TIMER_03)
```



```
FUNCTION:  TIME DELAY "OFF TO OFF"
STATEMENT:  B = ~TIMER (~A, ~A, TIMER_04)
```



Notes:

Appendix A

Message Format

Compiler errors result in specific error messages written to the .LIS file and a summary message written to SYS$OUTPUT. The format of error messages is as follows.

%CRISP-sevr-error   description

%CRISP	The facility name for the CRISP compiler. This name begins all error messages and is useful in searching for errors.
sevr	This one-letter code refers to the severity of the error. The levels of severity are as follows. **I - (INFORMATION) -** Compiler executes normally. An assumption has been made by the compiler to correct what appeared to be a typographic error. Compiled logic will execute properly. **W - (WARNING) -** Compiler executes normally. An assumption has been made by the compiler. The logic will execute, but it may not operate as planned. **E - (ERROR) -** Compiler executes normally except that no database object or MACRO files are produced. **F - (FATAL) -** Compiler cannot execute normally.
ident	The abbreviation of the message text. The message descriptions are alphabetized by this abbreviation.
text	This is a description of the error condition.

Compiler Messages

The following lists the compiler messages, defines the message type, explains the cause of the messages, and defines the user action that should be taken in order to correct the problem.

BADIDENT, A delimited string must follow IDENT;

Message Type: Error

Explanation: The IDENT statement must contain a string of up to 15 characters enclosed in quotes (").

User Action: Check that both quotes are present and that no characters other than space and tab appear between the semicolon and the first quote.

BADTITLE, A delimited string must follow TITLE;

Message Type: Error

Explanation: The TITLE statement must contain a string of up to 63 characters enclosed in quotes (").

User Action: Check that both quotes are present and that no characters other than space and tab appear between the semicolon and the first quote.

CGENERR, A code generation error occurred at line n

Message Type: Error

Explanation: An internal problem has occurred in the compiler.

User Action: Contact Square D Company, CRISP Automation Systems.

CTRCHRIG, Control character ignored

Message Type: Information

Explanation: An unsupported ASCII control character was located in the preceding statement. The supported control characters are horizontal tab and form feed.

User Action: Edit the source file and delete the invalid character.

EOFFOUND, End of file found before END;

Message Type:	Error
Explanation:	The last statement in a CRISP logic program must be an END statement.
User Action:	Edit the source file and add the END statement.

EXTACCIG, Extraneous accent mark ignored

Message Type:	Information
Explanation:	The grave accent (`) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTATSIG, Extraneous at-sign ignored

Message Type:	Information
Explanation:	The at sign (@) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTBKSFO, Extraneous backslash found

Message Type:	Information
Explanation:	The backslash (\) character is not valid in a CRISP logic program except within a delimited string or as a line continuation symbol. If used for line continuation, the backslash must be the last character on the line to be continued not including spaces, tabs, and comments.
User Action:	Edit the source file and delete the invalid character.

EXTCCBIG, Extraneous close brace ignored

Message Type:	Information
Explanation:	The right brace (}) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTDOLIG, Extraneous dollar sign ignored

Message Type:	Information
Explanation:	The dollar sign ($) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTLBSIG, Extraneous pound sign ignored

Message Type:	Information
Explanation:	The pound sign (#) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTOCBIG, Extraneous open brace ignored

Message Type:	Information
Explanation:	The left brace ({) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTPCTIG, Extraneous percent sign ignored

Message Type:	Information
Explanation:	The percent sign (%) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTPERFO, Extraneous period found

Message Type:	Warning
Explanation:	The period (.) character is not valid in a CRISP logic program except within a delimited string or as part of a floating-point constant.
User Action:	Edit the source file and delete the invalid character.

EXTQUMIG, Extraneous question mark ignored

Message Type:	Information
Explanation:	The question mark (?) character is not valid in a CRISP logic program except within a delimited string.
User Action:	Edit the source file and delete the invalid character.

EXTSCNFO, Extraneous semicolon found

Message Type:	Warning
Explanation:	The semicolon (;) character is not valid in a CRISP logic program except within a delimited string or as the delimiter after the keyword in a CRISP statement.
User Action:	Edit the source file and delete the invalid character.

EXTSPCFO, Extraneous underscore found

Message Type:	Warning
Explanation:	The underscore (_) character is not valid in a CRISP logic program except within a delimited string or within a variable name. It may not be used to begin a variable name.
User Action:	Edit the source file and delete the invalid character.

FOUND, Found "xxx**" when expecting ...**

Message Type: Error

Explanation: This message is generated during syntactical analysis of the source file when the entity (*xxx*) shown in quotes was found but it is not of an expected type. An example of this message is shown below where an underscore was intended but a minus sign was actually entered.

```
    31  NUMERIC;      XXX0_1, XXX-2, XXX0_3
%CRISP-E-FOUND, Found "-" when expecting a variable, comma, or EOLN
```

User Action: Correct the statement.

INVCONS, Constant value out of range for xxx

Message Type: Warning

Explanation: The initial value for the specified variable (*xxx*) is too large to fit in a variable of that type. For example, 40000 and -40000 are too large for a numeric variable. The initial value is ignored.

User Action: Use an appropriate initial value or declare the variable to be a Long.

INVDIMEN, Array dimensions must be >0 and <=32767

Message Type: Error

Explanation: The dimension for an array must be an integer constant that is greater than zero and less than or equal to 32767.

User Action: Correct the dimension value.

INVLOGINI, Invalid initial value for logical variable xxx

Message Type: Warning

Explanation: The initial value for a logical variable (*xxx*) must be either the keyword TRUE or the keyword FALSE. For array variables, two successive colons may be used to indicate an uninitialized (false) element in the array.

User Action: Correct the statement.

INVSUBSC, xxx **array subscript out of range**

Message Type:	Error
Explanation:	The statement (*xxx*) contains a constant array subscript that is outside the valid range for the array variable shown. Valid subscripts range from zero to one less than the array dimension (0 to n-1).
User Action:	Correct the subscript value.

LBLTOOLNG, Label name must be<= 30 characters

Message Type:	Error
Explanation:	Label names cannot be longer than 30 characters.
User Action:	Shorten the label name in the LABEL statement and any JUMP statements that refer to it.

MULTDEFV, xxx **is multiply defined**

Message Type:	Error
Explanation:	The name (xxx) shown in the message has been previously defined as either a variable or a label. In CRISP, these names must be unique.
User Action:	Change one of the names in all appropriate locations.

NOLABEL, The label xxx **was not defined**

Message Type:	Error
Explanation:	The label (xxx) shown in the message was not defined by a valid LABEL statement.
User Action:	Correct the spelling of the label in the JUMP statement or add the required LABEL statement in the desired location.

NOOBJPRO, No object produced

Message Type:	Information
Explanation:	Due to compilation errors, the database object (.DBO) and Macro-32 (.MAR) files were not written.
User Action:	Correct the other errors and retry the compile.

NOTARRAY, Illegal subscript, xxx is not an array

Message Type:	Error
Explanation:	A subscript was specified for the variable (xxx) shown that was not declared to be an array.
User Action:	Either remove the subscript or correct the variable declaration.

RESVDWRD, Reserved word xxx was found

Message Type:	Error
Explanation:	One of the CRISP reserved words (xxx) was found in a situation where it is not allowed. This might happen when using a reserved word for a variable name or if the semicolon delimiter is omitted following a statement keyword. Refer to Appendix D for a list of reserved words.
User Action:	Choose a different name for the variable or correct the statement syntax.

STRSZLRG, String size too large, truncated to 130

Message Type:	Warning
Explanation:	The explicitly declared size for a string variable exceeded the maximum of 130 characters. The compiler will treat this string as if it were declared to be 130 characters long.
User Action:	Correct the string size declaration.

STRSZSML, String size too small, set to 2

Message Type:	Warning
Explanation:	The explicitly declared size for a string variable was less than the minimum of 2 characters. The compiler will treat this string as if it were declared to be 130 characters long.
User Action:	Correct the string size declaration.

STRTRUNC, Initial string truncated to declared size

Message Type:	Information
Explanation:	The initial value for a string variable is longer than its declared size. The initial value will be truncated to fit the variable.
User Action:	Correct the string size declaration or shorten the initial value.

TICCBDTYP, "'" used with variable xxx, which is not a TIMER or COUNTER

Message Type:	Warning
Explanation:	The address modifier character (') was used on the displayed variable (xxx), which is not a timer or counter.
User Action:	Edit the source file and delete the invalid character.

TOOMNYINI, Too many initial values supplied, extra ignored

Message Type:	Warning
Explanation:	More initial values were supplied for an array variable than were declared by the dimension of the array.
User Action:	Delete the extra values.

UNDEFVAR, Found undefined variable xxx

Message Type: Error

Explanation: The displayed variable (xxx) was not defined in a valid declaration statement.

User Action: Correct the spelling of this instance of the variable or declare it in the variable declarations section.

VNAMTRNC, Variable name xxx truncated to 31 characters

Message Type: Warning

Explanation: A variable name (xxx) longer than 31 characters was used.

User Action: Shorten the variable name to valid length.

Memory Requirements

The CRISP compiler can require a large amount of virtual memory if the logic program being compiled is large. If the compiler aborts with the error condition LIB$_INSVIRMEM, it has run out of memory due to one of two reasons.

In most cases, the page file quota (PGFLQUO) in the system authorization file for the user account needs to be increased. You can confirm this by typing `SHOW PROCESS /ACCOUNT` on the terminal on which the compile was attempted. If the value for 'Peak virtual size' is very close to or equal to the PGFLQUO quota, then that quota should be raised using the AUTHORIZE utility. Note that you must log out and then log back in after the quota has been increased before the change will have any effect. A reasonable amount to increment this quota would be 2000 to 4000 (pages).

On systems with a small amount of physical memory (less than 10 MB), it is possible to run into the VIRTUALPAGECNT limit with a VERY large program. VIRTUALPAGECNT is a SYSGEN parameter which controls the maximum size of any process on the system. This value may be increased (carefully) by adding VIRTUALPAGECNT to the SYS$SYSTEM:MODPARAMS.DAT file and performing an AUTOGEN and reboot. It is recommended that you read the description of VIRTUALPAGECNT in the VMS System Generation Utility Manual before doing this.

ASCII Character Set

Decimal Value	ASCII Char	Mnemonic/ Explanation
0	^@	NUL
1	^A	SOH
2	^B	STX
3	^C	ETX
4	^D	EOT
5	^E	ENQ
6	^F	ACK
7	^G	BEL (bell)
8	^H	BS (backspace)
9	^I	HT (Horiz. Tab)
10	^J	LF (Line feed)
11	^K	VT (Vertical Tab)
12	^L	FF (Form Feed)
13	^M	CR (Carriage Rtn)
14	^N	SO (Shift Out)
15	^O	SI (Shift In)
16	^P	DLE
17	^Q	DC1 (XON)
18	^R	DC2
19	^S	DC3 (XOFF)
20	^T	DC4
21	^U	NAK
22	^V	SYN
23	^W	ETB
24	^X	CAN
25	^Y	EM
26	^Z	SUB
27	ESC	ESC (escape)
28	^\	FS
29	^]	GS
30	^^	RS
31	^_	US

Decimal Value	ASCII Char	Mnemonic/ Explanation
32		SP (space)
33	!	exclamation point
34	"	quotation marks
35	#	number sign
36	$	dollar sign
37	%	percent sign
38	&	ampersand
39	'	apostrophe
40	(	left parenthesis
41	)	right parenthesis
42	*	asterisk
43	+	plus
44	,	comma
45	-	minus (hyphen)
46	.	period (dot)
47	/	slash
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	colon
59	;	semi-colon
60	<	less than
61	=	equals
62	>	greater than
63	?	question mark

Decimal Value	ASCII Char	Mnemonic/ Explanation
64	@	commercial at
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	left bracket
92	\	back slash
93	]	right bracket
94	^	circumflex
95	_	underscore

Decimal Value	ASCII Char	Mnemonic/ Explanation
96	`	grave accent
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	left brace
124	|	vertical bar
125	}	right brace
126	~	tilde
127	DEL	delete

Notes:

CRISP Reserved Logicals

The CRISP Reserved Logicals are the first 16 Logical variables in the CRISP database and are reserved for use by the CRISP system. The function of these Logicals remains fixed regardless of how they have been declared. The standard names for these flags and a brief functional description are as follows.

Bit	Name	Description
1	NEW_DB *(page C-2)*	New CRISP database has been copied to the Running Database. (Should be initialized to TRUE.)
2	NEW_CLE *(page C-2)*	CRISP Logic Executive (CLE) has restarted.
3	ACTIVE *(page C-2)*	Indicates active CPU in redundant systems.
4	CACDSA *(page C-2)*	Disables argument type checking on logic calls.
5		Reserved
6		Reserved
7		Reserved
8		Reserved
9		Reserved
10		Reserved
11		Reserved
12	ICCDSA *(page C-2)*	Disables Inter-computer communications (ICC) transfers between redundant hosts.
13	PAUSE *(page C-2)*	Pauses logic at END;.
14		Reserved
15		Reserved
16		Reserved

CRISP Reserved Logicals (cont)

These CRISP Reserved Logicals must be the first Logicals declared in a CRISP/32 Application program and are declared as follows.

```
LOGICAL; NEW_DB:TRUE,NEW_CLE,ACTIVE,CACDSA,\
                  ,,,,,,,ICCDSA,PAUSE,,,
```

NEW_DB

The NEW_DB bit must be declared with an initial state of TRUE. It remains TRUE for the first pass of logic, after which it is set FALSE. Note that this variable is false even on the first logic pass if CRISP is started with the RESTORE option or if the logic is restarted individually.

NEW_CLE

The NEW_CLE bit is set TRUE when the CRISP Logic Executive (CLE) process has restarted and is recovering. This flag allows the application program to re-execute any statement that communicates with CLE (such as the keylink calls). After one pass of logic, this flag is set FALSE.

ACTIVE

The ACTIVE bit, is set TRUE by CRMON when the CPU is the active CPU in a redundant CRISP system. CRMON also clears the ACTIVE flag in all databases when a switchover causes this CPU to become idle. CRMON monitors and controls the Arbiter Board in a redundant system.

CACDSA

The CACDSA bit, when TRUE, disables argument checking on CRISP Function calls. Normally, the argument list is checked on each call to ensure that the variable specified as arguments are the correct type. If a problem is detected, the call is not executed and a message is printed on the CRIPS$TT device, identifying the call name and the argument (excluding cond) that is invalid.

ICCDSA

The ICCDSA bit controls ICC transfers for this database only. ICC transfers are disabled while it is true.

PAUSE

The PAUSE bit, when TRUE, prevents the scheduling of the CRISP logic. It also stops timer tick-down. This can be helpful when debugging the system.

Appendix D

Valid Characters

CRISP variable names and labels may be composed of from 1 to 31 characters as follows (the first character must be a letter). The case of the variable names and keywords is not significant.

- Letters (abcdefghijklmnopqrstuvwxyz)
- Numbers (0123456789)
- Underscore symbol (_)

The following ASCII characters are used in the CRISP/32 language.

`!`	Comment delimiter
`"`	String delimiter
`&`	AND operator
`'`	Second word of counter, timer
`( )`	Enclose expression or array dimension/subscript
`*`	MULTIPLY operator
`+`	ADD operator, unary plus
`,`	Separator
`-`	SUBTRACT operator, unary minus
`.`	Decimal point
`/`	DIVIDE operator
`:`	Variable initialization
`;`	Keyword terminator
`<`	LESS THAN operator
`=`	EQUAL TO operator, assignment equals
`>`	GREATER THAN operator
`[]`	String size and history initialization delimiters
`\`	Line continuation
`|`	OR operator
`~`	NOT operator
`^`	XOR operator

Reserved Words

The following words are reserved and cannot be used as a variable name or label.

CALL
CONSTANT
COUNTER
END
FALSE
FLOAT
IDENT
INTERMEDIATE
JUMP
LABEL
LET
LOGICAL
LONG
MESSAGE
NUMERIC
RECALL
RESTART
SET
STRING
TABLES
TIMER
TITLE
TRUE
R0, R1, ...through R15, AP, FP, SP, PC, IV, DV