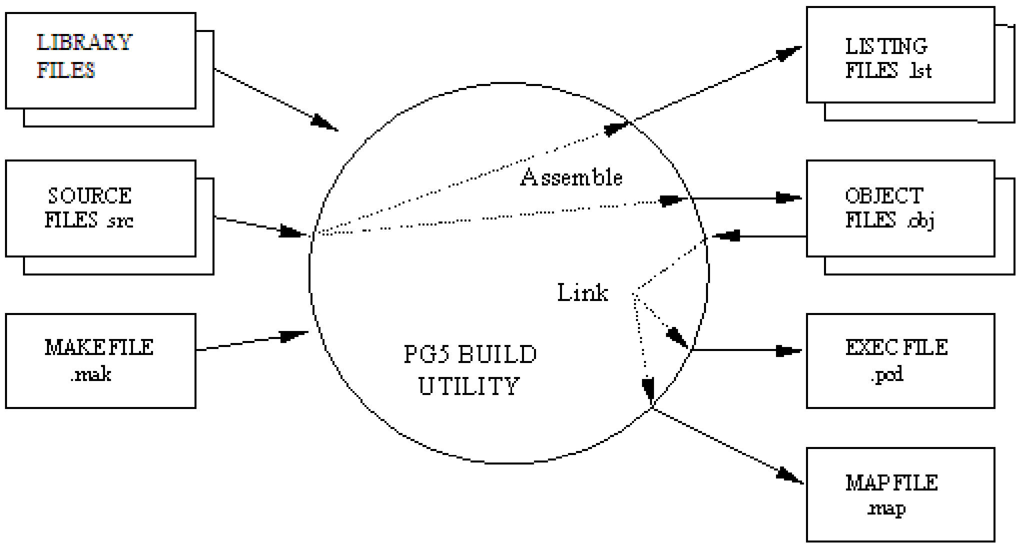
Saia PG5® Instruction List Language
Manual 26/733

1 Introduction

This document describes the Saia PG5 Instruction List language (IL), and the Saia PG5® Build Utility's messages and file formats.

The Build Utility - Assembling and Linking
The Saia PG5® Build Utility (S-Asm) processes one or more source files (.src) containing Instruction List code (IL), and creates a binary object file (.obj) and and optional listing file (.lst) for each source file. If there are no errors, it then links the object files together to produce a binary PCD-executable file (.pcd) and an optional map file (.map). The list of source file names, the .PCD file name and options switches are passed to the Build Utility in a make file (.mak). For new PCD models, a file called "PROGRAM.SPCD" is also created which contains the user program in a form that can be downloaded into the PCD's file system.



Notation
The following notation is used in the descriptions of instruction list statements in this document: Optional statements are shown enclosed in [square brackets], data descriptions are shown in italics, upper case characters must be entered as shown.
For example:

```
symbol EQU [type] value [;comment]
```

`symbol` is the symbol name, `EQU` must be entered as shown, `type` is optional, so is `;comment`.

Instruction format
Each instruction line has the form:

```
[label:] [mnemonic] [operand] [;comment]
```

Each field must be separated by one or more spaces or tabs as a delimiter, except for the comment field where the ";" character is the delimiter. Each line must end in a carriage return and/or linefeed character (e.g. Enter).

Instruction Presentation

Description	What the instruction does and its operands.
Format	Shows how the instruction is used and gives the type and range of each operand.

	An [X] after the mnemonic means that indexed addressing is possible by adding the optional X to the mnemonic (e.g. STHX, INCX). For indexed addressing, indexed operands are marked with "(i)".
Example	A typical example of the instruction.
Flags	Shows which Status flags are affected (ACCU, N, P, Z, E).
See also	A list of other instructions or topics which may be useful.
Practical example	Optional diagram and small program containing the instruction.

Typographic Conventions

`[ ]`	Square brackets enclose optional text or data. For example: [;comments] means that ";comments" is optional and need not be present.
`[X]`	An [X] after the mnemonic means that indexed addressing is possible by adding the optional 'X' to the mnemonic (e.g. STHX, INCX).
`(i)`	When indexed addressing is used, see [X] above, the indexed operands are marked with "(i)"
`< >`	Angle brackets enclose texts or expressions which should not be typed verbatim, but replaced by the relevant text or expression.
`|`	The "|" character means OR, e.g. [T|t] means an optional T or t can be entered, but not both.

1.1 Data Types

These are the data and block types used in IL programs.

Type	Description	Range	Notes
I	Input	0..8191	} I/Os share same addresses
O	Output	0..8191	} (Note 1)
F	Flag	0..8191/14335/16383	Volatile/Nonvolatile (Note 7)
T	Timer	0..450	Volatile, set to 0 at start-up (Note 2)
C	Counter	0..1599	Nonvolatile (Note 2)
R	Register	0..4095/16383	Nonvolatile (Note 3)
K	K constant	0..16383	
COB	Cyclic Organization Block	0..15	
XOB	Exception Organization Block	0..31	(Note 4)
PB	Program Block	0..299	
FB	Function Block	0..999	
SB	Sequential Block	0..31/95	(Note 5)
IST	Initial Step	0..1999/6999	(Note 5)
ST	Step	0..1999/6999	(Note 5)
TR	Transition	0..1999/6999	(Note 5)
X	Text	0..7999/8191	} Texts/DBs share same addresses
DB	Data Block	0..7999/8191	} (Note 6)
S	Semaphore	0..99	
STR	String	-	New in PG5 V2.1

In addition to the types, some attributes can be specified:

`R FLOAT`	Register containing a Motorola Fast Floating Point number (FFP)

`R IEEE`	Register containing an IEEE Floating Point number (single)
`F VOL`	Volatile Flag, required for dynamic address allocation so address is allocated from "Dynamic Volatile Flags" range.
`TEXT RAM`	Text in Extension Memory (RAM), required for dynamic address allocation so address is allocated from "Dynamic RAM Texts" range.
`DB RAM`	Text in Extension Memory (RAM), required for dynamic address allocation so address is allocated from "Dynamic RAM Data Blocks" range.

Inputs and Outputs

Inputs and Outputs are via interface modules which are plugged into the PCD. The address range of the module depends on which slot it is plugged in to.
Input states can only be read. Outputs can be turned on (set to 1 or High), and turned off (reset to 0 or Low), and their state can also be read.

Flags

Flags are 1-bit data which can be treated in the same way as Outputs, e.g. they can be set or reset, and their state can be read. See Note 7 below.
Tip: If you need a large number of Flags but don't need to access them very fast, think about using bits of Registers or bits in DB elements. In IL these can be accessed easily using Macros.

Timers and Counters

Timers and Counters are unsigned 31-bit values (0..2'147'483'647 in decimal), they can hold only positive values. Timers and Counters share the same address range from 0..1599. The number of Timers is defined by the instruction `DEFTC`. The default value is 32 Timers from addresses 0 to 31, and 1568 Counters from addresses 32 to 1599)
The only difference between a Timer and a Counter is that a Timer is decremented according to the timebase defined by the instruction `DEFTB`, The default value is 1/10th sec (100ms). The DEFTC and DEFTB instructions are generated from the device's Build Options in the Project Manager.
When a Timer or Counter contains a non-zero value its state is High (H or 1), when its content is zero its state is Low (L or 0)

Registers

A Register is a 32-bit data store which can hold data in binary, decimal, hexadecimal, or floating point or IEEE units. You can perform arithmetic operations on Registers, or transfer data to or from: Inputs, Outputs, Flags, Timers, Counters, DBs or other Registers. See Note 3 below.

Constants

The IL language supports integer constants (13, 16 or 32 bits), 32-bit floating point values (Motorola FFP or IEEE Float), or 64-bit IEEE Double values. See Numeric Constants. For instructions which can have a data address or a constant as an operand, use the K data type. The K type of constant is restricted to 14 binary bits, for an example, see ADD.

Texts

Texts are strings that can be stored in the PCD for transmission over a communications line, or sent to a display terminal.

Data Blocks (DBs)

A Data Block is block which contains an array of 32-bit data vakues, which can be transferred to and from Registers, Timers and Counters.
Texts and DBs share the same addresses. 0..3999 are in Text/DB memory which may be read-only Flash or EPROM memory. Texts/DBs 4000..8191 are in Data Memory (also known as Extension Memory), which is always RAM (read-write). In some PCD types this partition can be defined using the Project Manager's Build Option "First writeable Text/DB number".

Strings

Strings are new in version 2 of the PG5, see Strings, STR and @STR( ) for more details.

NOTES

1. The max. number of I/Os depends in the PCD type. Each module's I/O address depends on the module's slot position. See your PCD's hardware manual for details.
2. Timers and Counters share the same address space. The low addresses are always Timers, the rest are Counters.
   The number of Timers is defined by the Project Manager's Build Options (or by DEFTC). Timers are 'volatile' and are all set to 0 on start-up.
   Counters are 'nonvolatile', their values are not lost when the PCD is powered off and on, except for PCD types without a backup battery.
3. Old PCD models have Registers 0..8191. New systems (PCD3, PCD2.M480, PCD1.M2xxx etc) have Registers 0..16383.
   For FW versions before 1.20.0, only Registers 0..8191 can be used for indirect addressing (see TFRI etc).
   FW versions from 1.20.0 can use all Registers for indirect addressing, providing the Build Option "Use 16-bit addressing" is Yes.
   All Registers are nonvolatile.
4. XOBs have fixed purposes according to the XOB number, see XOB.
5. Graftec: New PCD systems support SBs 0..95, and ST/TRs 0..5999. Older systems support SBs 0..31 and ST/TRs 0..1999.
6. Texts and DBs 0..3999 are in Text/DB memory, which may be read-only Flash or EPROM memory.
   Texts/DBs 4000 and above are always in writeable RAM memory (data memory). New systems support Texts/DBs 0..8191.
   The PCD1 has up to 5999, and older PCD2s support up to 5999. For other PCD models refer to the hardware manual.
7. Old PCD models have Flags 0..8191. New PCD systems from firmware version 1.14.3 support Flags 0..14335.
   Flags 0..16383 are supported by PCDs with FW version 1.20.0 or later, providing the Build Option "Use 16-bit addressing" is Yes.

1.2 Condition Codes [cc] and Arithmetic Status Flags

Arithmetic Status Flags

The arithmetic status flags are affected mostly by the Integer and the Floating Point instructions which are set according to the result of each instruction.
The Error flag is set High by any instruction which is executed with invalid data or fails in some other way.

P	Positive	High if result of an arithmetic instruction is positive
N	Negative	High if result of an arithmetic instruction is negative (the P flag is always the inverse of the N flag)
Z	Zero	High if the result of an arithmetic instruction is 0
E	Error	High if an instruction fails to execute, for example on overflow, underflow or conversion error

Accumulator

The Accumulator (ACCU) is set High or Low (1 or 0) mostly by the Bit instructions.
It can be set to a specific state, or to the state of an arithmetic status flag, using the ACC instruction.
The ACCU is often used to control a sequence of bit instructions where each instruction depends on the result of the previous one. This normally begins with a start instruction, e.g. STH, and ends with an action instruction, e.g. OUT. The intermediate result of each bit instruction is stored in the ACCU. The final ACCU state is the result, which can be written to a Flag or Output.

NOTE

Many instructions are ACCU dependent, and are executed only if the ACCU is High (1). This is indicated in the instruction description.

Condition Codes [cc]
Condition codes [cc] define the Status flag states which allow execution of the instruction. If the condition is false, the instruction is not executed. For example a jump instruction JR Z will not be executed unless the Zero status flag is High (1).

Code	**Description**
blank	No condition code
`H`	If Accumulator = H (1)
`L`	If Accumulator = L (0)
`P`	If Positive flag = H (Negative flag = L)
`N`	If Negative flag = H
`Z`	If Zero flag = H
`E`	If Error flag = H
`C`	Complement used with the ACC instruction only

1.3 Symbol Names

Symbol names are names which are assigned to PCD data like Inputs, Outputs, Flags, Registers, Texts, or to code blocks like COBs, PBs and FBs.

Symbols can be up to 80 characters long, and are not case-sensitive unless they contain accented characters. `MotorOn` is the same as `MOTORON`, but `FÜHRER` is not the same as `führer`.

Symbol names are assigned types and values using EQU or DEF declarations, and also the more recent LEQU, LDEF an PEQU declarations.

Symbols can also have group names, defined by the $GROUP directive, which adds a prefix to each symbol. Groups can be used to define unique symbol names if similar code is repeated several times, for example inside a MACRO which defines public symbols and is used several times in a program, or for an instance of an FBox.

These are the rules for symbol names:

- Symbols cannot begin with a digit (0-9), and must be two or more characters in length.
- Symbols must use the standard Window's ANSI character set. These characters allowed:
  A-Z
  a-z
  0-9
  _ (underscore)
  Accented characters with ANSI character codes, but see the NOTES below.
- Symbols cannot begin with an underscore "`_`", this is reserved for internal symbol names.
- Reserved Words cannot be used as symbol names.
- The assembler pre-defines some internal symbols, see Pre-defined Symbols, new symbols cannot be defined using these names.
- Symbols can have group names, either using `$GROUP` statements or by using a dot '.' to separate each group name, e.g. `Group0.Group1.Symbol0`s
- Group names starting with one character, e.g. "S." are reserved for system symbols and should not normally be defined by user programs in case there are conflicts.
- Sub-group names may be instruction mnemonics, e.g. TEST, but only if the symbol name is defined in full as shown below, because the $GROUP directive does not allow the group name to be a reserved word.

```
Group0.Test.Symbol EQU R
```

NOTES

- In some cases it may be advisable to avoid using accented characters in symbol names. S-Asm supports them, but they can cause problems if a program file is transferred to another PC with a different operating system (Windows 2000/NT/XP/Vista/7 etc) or with different language support installed. Some characters are not translated properly by the operating system, and the file will not assemble.
- The DOS-based PG3 and the 16-bit PG4 used the old "OEM character set", which may not be compatible with the PG5 if accented characters have been used. The current version of the PG5 uses the ANSI character set for program files. Fupla programs will be converted automatically from the OEM to the ANSI character set. IL programs can be converted using the IL Editor's "Convert from OEM to ANSI" command.
- From PG5 version 2.0.200, Fupla supports Unicode and has a selectable ANSI code page for symbol translation.

1.4 Scope of Symbols

A symbol's "scope" defines its visibility to other blocks and files, and the lifetime of the data that the symbol references. IL programs can have symbols with several scopes.

The Symbol Editor has a column for the scope of a symbol, and it supports Public, External and Local (local to the file). The Symbol Editor does not support "Local to the block" or "Temporary data" scopes, which can be used only from IL programs - see below.

Public

Public symbols are declared with PUBL or PEQU. They can be referenced from any file in the program. Public symbols also be exported, renumbered etc. Unless explicitly declared public, symbols have scope only within the file in which they are defined. They cannot be referenced from other files. Public symbols can also be declared in global $include files, see "Global symbol files" below.

Tip: Do not make symbols Public unless they really will be accessed from other files, or you need the features which are only available to Public symbols.

NOTE

All data and block numbers R T C I O F COB XOB PB FB SB IST ST TR TEXT DB are global, even if their symbols (if used) are not explicitly declared PUBLic.

For example, R 100 ca be accessed directly from any file or part of the program, without using a symbol name.

This is a common source of programming errors because the user may accidentally access the same data using different symbol names from more than one place in the program.

This can be checked by using SPM's Build Option "Warn on symbols with same type and address".

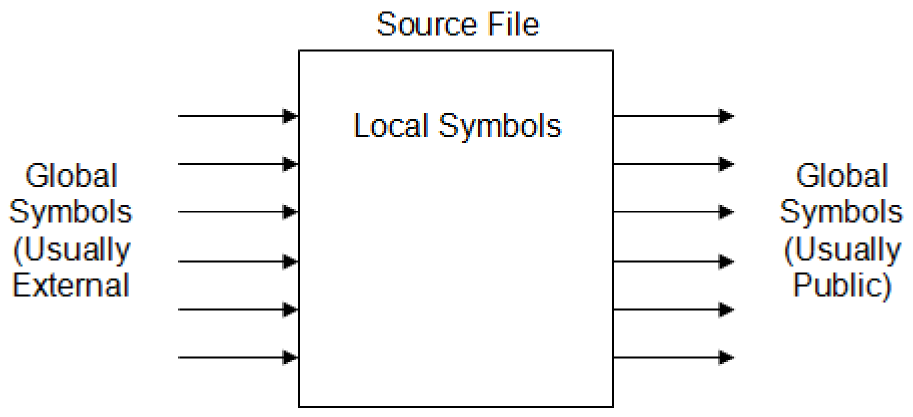
WARNING

If a block uses fixed data addresses, and the data needs to be retained between each block call (i.e. it is not re-initialized every time), then you **cannot** call the block from more than one COB (more than one *task*) because the data will be valid for only one task, not for both, unless the code has been specially written to support this. This is another common source of programming errors. Blocks which are shared by more than one task must use different static data for each task, either by using Register Indirect addressing, by passing the addresses as FB parameters, or clever use of the Index Register.

External

A symbol whose actual type and value is defined in another file can be declared as an external symbol using EXTN. In Symbol Editor, use the "External" scope.EXTN declarations can be placed in the referencing IL file, or in an include file which defined the "interface" to another IL file.

Symbols which are declared Public are know as "Global" symbols. An IL file's global symbols can be thought of as the "interface" to the IL file, just like an FBox's inputs and outputs. The local symbols are only accessible from inside the same file.


Source File
Local Symbols
Global Symbols (Usually External
Global Symbols (Usually Public)

Local to the file
Symbols which are declared with DEF or EQU (without PUBL) can be used only in the file in which they are defined.
In Symbol Editor, use the "Local" scope.

Local to the block or macro
Symbols declared with LEQU and LDEF are local to the block or marco in which they are defined. They cannot be referenced from outside the block or macro, and cannot be made Public.
This scope is not supported by the Symbol Editor.

Local symbols which can be re-defined (DEF)
Normally a symbol can have only one declaration, otherwise a "multi-defined symbol" error occurs. But often you may want to re-define a symbol's value so it can be used as a reference or loop counter. This can be done using DEF. For example, if you wanted to increment a symbol's value each time a macro was called, you could declare the macro like this:

```
MyMacro MACRO
RefCounter DEF RefCounter + 1
...
ENDM

RefCounter DEF 0
MyMacro()    ;increments RefCounter
MyMacro()    ;increments RefCounter
```

Temporary data
Register and Flag data which is needed only while a block executes does not need to use the normal Registers or Flags. Insetad, temporary Regsietrs or Flags can be used, which disappear at the end of the block. These are for use as workspace data.
(For those familiar with high-level languages like "C", temporary data would be on the "stack", whereas the normal Registers and Flags can be thought of as being on the "heap".)
Temporary data are declared using TEQU. It can only be declared inside a block (COB, PB, FB, ST or TR).
Symbols declared with TEQU cannot be accessed from outside the block - they only exist while the block is running.
Temporary data is not supported by the Symbol Editor.

Global symbol files
Another way to define public symbols is to put them in a global include file. This is how it was done in PG5 V1.x, but PG5 V2 has introduced a safer way to do it.
Global include files are still supported by PG5 V2, but only for supporting programs written with PG5

V1.x which used the Global.sy5 file, or for symbol files generated from Excel files or other code generators.
In PG5 V2, we encourage you to use the Public/External mechanism - which is now available in Symbol Editor, and declare the Public symbols in the files which create them, and the External symbols in the files which reference them.
This keeps the symbols in the files which create them, instead of having the symbols defined in different file. The file can then be copied or moved without losing the symbol definitions - better "encapsulation".
Global symbols files are not compatible with the new "background build" because any changes to the file means that ALL the program's files must be re-assembled and linked. This will be very slow.

Tip: We recommend that you only use global symbol files for symbol names which will not be regularly changed. Do not use global symbols files unless they are really necessary. Some old PG5 V1.x users put all their symbols into the old Global.sy5 file. This is not recommended anymore because it is not compatible with the new background build, any changes to a global include file means that ALL files must be compiled, assembled and linked, which can be a long procedure.

Forward and backward references
EXTN, EQU and PEQU symbols can be both forward and backward referenced, they can be declared anywhere in the source file, and referenced from anywhere in the file - unless they are used in $IF statements where they must be defined before they are used. Symbols defined with DEF or LDEF have scope from the definition point to the end of the source file (unless re-DEFined), allowing backward references only.

Scope of Labels
Labels (symbol names for program line numbers) are usable only within the code block or macro in which they are defined, they are local to the block or macro.

1.5 Typed Symbols

When a symbol is EQUated, DEFined or declared as external EXTN (or PEQU LDEF, LEQU, GDEF, GEQU), a data type is normally assigned to the symbol.

If symbols are given a type then the type is checked whenever the symbol is used and provides added security. It is invalid to use a symbol with an invalid type in an instruction, or to mix symbols of different types in an expression. PUBLic symbols retain their type information which is checked by the Linker in the same way.

If a symbol has a type, it is not necessary to use the type in the instruction, but if a type is used then it must match the symbol's type. For example:

```
INPUT EQU I 1  ;declare "INPUT" as Input 1
...
STH   INPUT    ;same as "STH I 1"
STH   I INPUT  ;same as above, but symbol "INPUT" *must* be an Input
STH   F INPUT  ;eerror! Type "F" does match "INPUT" symbol's type
```

(Note: if you use S-Edit with the Symbol Editor, it will automatically remove the unwanted type when S-Edit processes the line.)

Permitted symbol types are (see Data Types for address ranges):

Type	Description
`I`	Input
`O`	Output
`I|O`	Input or Output (both use the same numbering)

`F`	Flag
`R`	Register
`T`	Timer
`C`	Counter
`T|C`	Timer or Counter (both use the same numbering)
`K`	Constant (13-bits unsigned)
`COB`	Cyclic Organization Block
`XOB`	Exception Organization Block
`FB`	Function Block
`PB`	Program Block
`SB`	Sequential Block
`ST`	Step (or Initial Step)
`TR`	Transition
`SEMA or S`	Semaphore (for LOCK and UNLOCK) (obsolete)
`TEXT or X`	Text
`DB`	Data Block
`DBX`	Extended Data Block
`IB`	Information Block
`=`	Function Block parameter number (not for EXTN symbols)

A symbol can also have type "label" if it is declared as a label.

The symbol's type appears in the TYPE field of the cross reference list in the listing file.

For dynamic Flags (without an address), there is also a VOL attribute which declares the Flag as 'volatile' so that its address will be assigned from the volatile flags segment (see Software Settings), e. g.

```
    MyFlag EQU VOL F   ;a volatile flag
```

For dynamic Texts and DB addresses, there is a RAM attribute so that the address will be selected from RAM Text or RAM DB segments (see Project Manager's Build Options), and is not stored in Flash or EPROM memory which is read-only, e.g.

```
    MyText EQU RAM TEXT ;this is a RAM text
```

Tip: New PCD types have a Build Option which defines the first writable Text/DB.

1.6 Numeric Constants

The default base for numeric constants is decimal. All constants are stored as 32-bit signed integers (one Register), except Double with is 64 bits (2 Registers).

The following types of numeric constants are available:

Decimal constants
Decimal values have the range -2'147'483'648 to +2'147'483'647 (signed 32-bit).

Binary and hexadecimal constants
Binary or hexadecimal bases can be used by post-fixing the number with a base indicator character:

`Y`, `y` or `Q`, `q`	Binary, e.g. `1001Q`, `11111111q`
`H` or `h`	Hexadecimal, e.g. `0FFH`, `07fh`

A hex value must always begin with a digit (0..9), otherwise it could be interpreted as a symbol if it begins with A..F.
Binary and hex constants have the range 0 to 0FFFFFFFFH.
Note that 0FFFFFFFFH is -1 decimal and 80000000H is -2147483648 decimal.

Character constants
These can be entered by enclosing the characters in single quotes, one to four characters can be

entered.
Each character uses 8 bits, so 4 characters fills a 32-bit integer,
e.g. `'A' 'ab' '?' 'abcd' 'ƒ'`
Decimal values for non-printable characters can be defined inside angle brackets, e.g. `'<10><13>'`.
To enter the `< >` and `'` characters, enclose these in angle brackets too, e.g. `'<<><>><'>'`.

K Constants

These are any decimal, binary or character constant which fits into 13 binary bits (range 0..3FFF hex, 0..16383). These are used in Integer instructions where the operand must have a data type (IL operands are 16-bit values and the upper 3 bits define the data type, leaving 13 bits for the value).

Floating point constants, "Motorola Fast Floating Point" FFP

Floating point values must contain a decimal point '`.`', and/or an 'E' (or 'e') followed by an exponent, e. g. `.23, 1.234, 1E10, 1e-10`.
Floating point constants cannot be used in expressions, IL expressions doing arithmetic operations with floating point values will generate an error. The range for FFP numbers is:

`+5.42101E-20 .. +9.22337E+18` (6 significant digits)
`-2.71056E-20 .. -9.22337E+18`

IEEE floating point constants, Float and Double (new)

Recent PCD firmware now supports IEEE Floating Point values, and contains instructions for processing these formats. Float (32-bit) and Double (64-bit) values are supported. A Float values uses a single Register, a Double value uses two consecutive Registers. To declare an IEEE float value, terminate the number with the letter `I` (without the `I` it is assumed to be a Motorola Fast Floating Point value).

```
    Symbol EQU 1.2I
```

Double values cannot be assigned directly to symbols, instead they must be loaded into a Register using the @DFPHI( ) and @DFPLO( ) operators, which return the upper and lower 32-bits of the 64-bit double value. These operators can also be used to convert FFP and Float values to double - in the example below, Symbol can be an IEEE float or Motorola Fast Floating Point value.

```
    LD      R  0
            @DFPHI(Symbol)  ;the upper 32 bits of the double value
    LD      R 1
            @DFPLO(Symbol)  ;the lower 32 bits of the double value
```

The range for Float is: +/-1.1754943e-38 to +/-3.4028234e+38 (7 significant digits)
The range for Double is: +/-2.2250738585072014e-308 to +/-1.7976931348623158e+308 (15 significant digits)

$ constants

'$' is assigned the value of the program line offset from the start of the current code block (COB, XOB, PB, FB, SB, IST, ST or TR). It can be used for creating a jump label for relative jumps (JR). $ cannot be used inside a $INIT or $xxxSEG segment. $ constants can also be made PUBLic, whereas labels cannot because labels are always local to the block in which they are defined.
For example:

```
    Label EQU $     ;offset from start of block
    ...
    LD      R 0
            Label
```

Tip: You can define labels directly using a colon, so $ is not really needed anymore.

```
Label:
    ...
    JR      Label
```

1.7 Time Constants (for loading Timers)

Timers are decremented at a rate defined by the DEFTB instruction (Define TimeBase). Timers are loaded with the number of "ticks" whose duration is defined by DEFTB.

The values loaded into Timers will therefore need to be changed if the DEFTB timebase is changed. To overcome this problem, the "time" data type can be used to declare Timer load values. If a time value is used, then the linker calculates the actual Timer load value according to the DEFTB timebase – changing the DEFTB timebase adjusts all the "time" values.

Time (duration) values can be in seconds or milliseconds. The maximum time that can be stored in a time value is 2147483 seconds (24.8 days).

Format: `T#nnnS|MS`

where:	`T# (or t#)`	Introduces the time data type
	`nnn`	The time value in seconds or milliseconds, range 10..2147483647 milliseconds, 1..21483 seconds
	`S|MS`	S = value is in seconds, MS = in milliseconds

Examples:
```
DelayTime    EQU   T#100MS       ;100 milliseconds
OneDay       EQU   t#86400s      ;86400 seconds
```

NOTE
The Timer load values calculated by the linker are rounded up to the next DEFTB tick, DEFTB defines the resolution of the Timers. For example, for DEFTB 100 (100 x 10ms = 1000ms), the lowest Timer value would be 1000ms (1 second), therefore "T#10ms" would be rounded up to 1000ms. For a DEFTB value of 1 (1 x 10ms) the lowest timer value would be 10ms for "T#10ms", which produces a Timer load value of 1.

1.8 Labels

Description
Labels are symbol names given to locations in a program (program lines), which are used as destinations for jump instructions or to provide debug information.

Characters allowed in labels are the same as those of symbols.
Labels can appear anywhere in the source file, but should be within a code block (COB, PB etc.), and must not be inside a multi-line instruction.

The value assigned to a label is its offset within the code block where it is defined.
All labels are local to the block in which they are defined, the same label can be used many times in a the same source module, providing it is always in a different block.
Jumps to labels defined in another block are not allowed.

Labels cannot be public, since all labels are local to the block in which they are defined, and jumps between source modules are not allowed.
'$' can be used to create untyped symbols containing the offset into the code block, which may be made public or can be referenced from another block (by the LD or RCOB instructions).

Example
```
      PUBL LAB1         ;LAB1 is declared public
      LAB1 EQU $        ;LAB1 is offset from start of block
      ...
LAB2: STH  I 0          ;LAB2 is a label for a wait loop
      JR   L LAB2
      ...
```

1.9 Texts TEXT or X

__Description__
Texts are arrays of characters, entered as strings enclosed in double quotes `"..."`. Texts can consist of one or more lines of text, each line must be opened and closed with double quotes.

Depending on the PCD type, some Texts are writable and some Texts are read-only. Writable texts can be overwritten, but cannot have their length changed at runtime. Writable Texts are in the Data Segment (also known as Extension Memory). Read-only Texts are stored in the Text Segment. For older PCDs the partition between these segments is fixed, Texts 0..3999 are read-only and Texts 4000 an above are writeable. Some recent PCD types allow the partition to be adjusted using Project Manager's Build Option "First writeable Text/DB number".

Texts 0..3999 are stored in the standard NUL-terminated string format - they must contain a NUL character (00 in hex) at the end, and **only** at the end. Texts 40000 and above can contain NUL characters anywhere, so with a bit of clever programming these can be used to support variable-length Texts.

The maximum number of characters allowed in a single Text is 3072 characters (3K).

Any character can be entered into a Text, except Texts 0..3999 do not allow the NUL character (numerical code 0) because this is used as the end-of-text character.

Symbols can be used in Texts, so you do not have to use absolute addresses, see Using Symbols in Texts.

__Format__
```
TEXT number [[length]]   "text line 1"    [;comment]
                         "text line 2"
                         ...
                         "text line n"
```

The Text number and length can be any expression or symbol combination. The length is an optional text length, which must be enclosed in square brackets, e.g. `[10]`, `[MaxTextLen]`. The text length defines texts for use with the PUT and GET instructions, which can copy blocks of Registers to and from Texts. If the length is present, then text in double quotes can be omitted, and the text is filled with space characters. If both the length and text are defined, the text is padded with space characters up to the given length.

__Examples__
```
ONE EQU TEXT 1
TWO EQU X 2               ;X and TEXT (data types) are the same

TEXT 0 [100]              ;Defines TEXT 0 as 100 spaces
TEXT ONE [5] "123"        ;Defines TEXT ONE as "123  "
TEXT TWO "123  "          ;Text TWO is the same as text ONE
```

__Special characters__
To place the `<`, `>` and `"` characters in a text they must be enclosed in angle brackets: `<<>`, `<>>`, `<">`
. In fact, the character value between angle brackets can be any expression, even containing user-

defined symbols, providing the value produced is in the range 0..255. Special characters can be entered as ASCII codes in decimal, or as standard ASCII mnemonics, enclosed in angle brackets, e. g. `"<CR><LF>"` or `"<10><13>"`.

The ASCII mnemonics are:

Mnemonic	Dec	Hex	Mnemonic	Dec	Hex
`<NUL>`	0	00H	`<DLE>`	16	10H
`<SOH>`	1	01H	`<DC1>`	17	11H
`<STX>`	2	02H	`<DC2>`	18	12H
`<ETX>`	3	03H	`<DC3>`	19	13H
`<EOT>`	4	04H	`<DC4>`	20	14H
`<ENQ>`	5	05H	`<NAK>`	21	15H
`<ACK>`	6	06H	`<SYN>`	22	16H
`<BEL>`	7	07H	`<ETB>`	23	17H
`<BS>`	8	08H	`<CAN>`	24	18H
`<HT>`	9	09H	`<EM>`	25	19H
`<LF>`	10	0AH	`<SUB>`	26	1AH
`<VT>`	11	0BH	`<ESC>`	27	1BH
`<FF>`	12	0CH	`<FS>`	28	1CH
`<CR>`	13	0DH	`<GS>`	29	1DH
`<SO>`	14	0EH	`<RS>`	30	1EH
`<SI>`	15	0FH	`<US>`	31	1FH
			`<DEL>`	127	7FH

Examples using special characters

```
TEXT 100 "<13><10>"          ;THIS IS CARRIAGE RETURN, LINE FEED
TEXT 101 "<CR><LF>"          ;SAME AS TEXT 100 ABOVE
TEXT 102 "<">Hello world!<">"   ;TEXT IS "Hello world!"

EndOfText EQU 2
TEXT 103 "<EndOfText>"       ;SAME AS "<2>"
TEXT 104 "<EndOfText+1>"     ;SAME AS "<3>"
```

NOTES

- Texts and Data Blocks share the same numbering. For example, if TEXT 10 is defined, then DB 10 cannot be defined, and vice versa.
- Texts 0..3999 are stored in Text memory, which can be RAM, EPROM or Flash EPROM, depending on teh PCD type. Texts 4000..8191 are only available on a PCD with Data (Extension) memory, which is always battery-backed RAM. The PCD2.M1xxx supports only Texts/DBs 0..5999.
- For Texts 0..3999, the first character of a text CANNOT be <253>, <254> or <255> (0FDh, 0FEh or 0FFh). These characters are reserved to indicate that the text is in a special format (binary LAN text or Data Block).
- The PCD2.M480, PCD3 and all recent PCD models (NT systems) support Texts/DBs up to 8191.
- An empty text (`""`) does not create a text in a .PCD file, but it does define the Text number for the assembler and linker, thus preventing that text from being re-defined. However, the PCD treats an empty text as though the text doesn't exist.
- The $SASI..$ENDSASI directives can be used to delimit Texts which are to be specially processed by the assembler.

1.10 Using Symbols in Texts

Texts can also contain references to symbols. The symbol's value or address and optionally the type of the symbol is inserted into the Text. The symbol is written outside the Text which is in double quotes, and must be separated from this or other symbols by a comma.

After the symbol, an optional field width and prefix type can be given.

For example:
`TEXT MyText "The value of Symbol3 is: ", Symbol3, "<CR>"`

Symbols in texts have this format:
`symbol[.[[-][0]width][t|T|c|C|s|s]]`

`symbol`	The symbol name. This can actually be any expression which includes a symbol, for example: `MotorOn+100`, `IOBase+Offset` etc. Note: Symbols with Floating Point or IEEE values are not supported.
`.`	The dot immediately after the symbol indicates that a field width and/or prefix is present.
`-`	If the width (see below) begins with - (a minus sign) then the displayed data is left-justified in the field. The default is right-justified.
`0`	If the width begins with a 0, leading zeros are inserted to pad the field to width characters. If width is preceded by a - sign, the data is left justified in the field and no leading zeros are inserted, even if the width begins with 0.
`width`	The field width, the number of digits or spaces for the number (1..20). If the number is longer (e.g. width=3, number=1234), then the width is automatically increased so the number is not truncated.
`t` or `T`	Optional prefix type t or T. If t, the value is prefixed with the symbol's type in lower case (o, f, r, pb, xob etc.). If T, the symbol's type is in upper case (O, F, R, PB, XOB etc.).
`c` or `C`	Indicates that the value is to be inserted into the text as an ASCII character. Only the least significant 8 bits of the value are used. e.g. `CharSym EQU 'A'` `Text 100 "Character is ", CharSym.C ;"Character is A"`

`s` or `S`	Inserts the symbol name into the text rather than the type/value of the symbol.
	`Symbol4 equ R 123`
	`TEXT 100 "Symbol name is: ", Symbol4.s`
	This is particularly useful in Macros, where the symbol can be a macro parameter.

These additional formatting characters are for use in IBs and DBXs only (not in Texts):

`d` or `D` Inserts the symbol's media type and address as a decimal ASCII number with the same format as a PCD operand.

```
Reg1 equ R 1
Flg2 equ F 2
DBX 0
@0:                  ;for variable length texts
Reg1.h, " ", Reg1.d  ;"8001 32769"
Flg2.h, " ", Flg2.d  ;"4002 16386"
EDBX
```

`h` or `H` Inserts the symbol's data type and address as a hex ASCII number with the same format as a PCD operand, see example above.

More examples of texts containing symbol references:

```
Flag    EQU F 123
Output  EQU O 32
Reg     EQU R 999
Char    EQU 'ABCD'
Block   EQU DB 100

TEXT 0 "$", Flag.04T          ;"$F0123"
TEXT 1 Flag                   ;"123"
TEXT 2 "DIAG:", Output.T, ",", Reg.T
                              ;"DIAG:O32,R999"
TEXT 3 "55:", Flag.T, "-", Flag+7, ":", Output.T, "-", Output+7
                              ;"55:F123-130:O32-39"
TEXT 4 "FLAG NUMBER: *", Flag.-8, "*"
                              ;"FLAG NUMBER: *123     *"
TEXT 5 "Char is ", Char.c     ;"Char is D"
TEXT 6 "Block is ", Block.T   ;"Block is DB100"
```

Texts can also contain formatted absolute addresses. This is useful in Macros so the parameter can be either a symbol or an absolute address.
Note that there must be a space between the type and the address, e.g. use R 100 not R100 (which would be interpreted as a symbol).

For example:	`TEXT TOTO "ABCDE$", R 100.04T`
generates	`TEXT TOTO "ABCDE$R0100"`

Symbols in SASI texts

```
DiagFlags EQU F 500
DiagReg   EQU R 4095

    XOB     16
    SASI    1
            3999

TEXT 3999 "UART:9600,7,E,1;MODE:MC0;"
          "DIAG:", DiagFlags.T, ",", DiagReg.T, ";"
```

This creates the text:
`"UART:9600,7,E,1;MODE:MC0;DIAG:F500,R4095;"`

1.11 Data Blocks DB

Description
A "data block" is a block which contains and array of 32-bit data values.

The values can be transferred to and from Registers, Timers and Counters. The PUT instruction transfers data from Registers into a Data Block, and the GET instruction transfers all the data from a Data Block into a block of Registers. The number of Registers copied depends on the DB length, which is defined in the DB declaration. The TFR (TRansFer) instruction transfers single elements to or from a DB.

DBs numbered 0..3999 can hold up to 383 values (0..382). these are stored in the Text memory segment, which may be Flash or EPROM, so they may be read-only. Access to these DBs is much slower than DBs 4000..8191.

DBs numbered 4000..8191 are always in RAM memory (Extension or "Data" Memory), and can contain up to 16384 values (0..16383). Access to these DBs is much faster than DBs 0..3999.

Because DBs 4000 and above are in RAM, their data will be lost if the battery fails or if the PCD does not have a memory backup battery. The SYSWR and SYSRD instructions can be used to save and restore these DBs (and the Texts) to/from flash memory, see function codes 3000..3002 and 3100..3102.

Format

```
DB number [[length]] [value1 [, value2]...]
```

The Data Block `length` must be given, enclosed in square brackets, e.g. `DB 10 [100]`, followed by an optional list of initial values. If the length is empty `[]`, then the length is determined from the number of values. Either the length and/or one or more values must be given. If only the length is supplied, the data is initialized to zeros. For DB 0..3999 length can be 1..383, for DB 4000..8191 the length can be 1..16384.

Values can be decimal, hex, floating point or ASCII, or can be any expression or symbol combination producing an integer or floating point value.

Examples

```
DB 100 [10]             ;DB 100 HOLDS 10 ITEMS,
                        ;ALL SET TO 0
DB 101 [4] 1,2,3,4.1    ;DB 101 HOLDS 4 ITEMS,
                        ;SET TO 1, 2, 3 AND 4.1
DB 102 [4] 1,2,3        ;DB 102 HOLDS 4 ITEMS,
                        ;SET TO 1, 2, 3, 0
DB DATA[LEN] VAL1,,,4H  ;DB DATA HOLDS "LEN" ITEMS,
                        ;SET TO VAL1, 0, 0, 4
```

Notes

- At present, data blocks 0..3999 are stored in "text memory". If the first character of a text is 0FFh (255), then it is treated as a data block. The data is encoded in a special way, so that NUL (0) bytes do not occur in the text. Data blocks 4000..7999 are stored in "data memory" in binary, and are not encoded. These can contain the NUL character.
- If the PCD contains flash or EPROM main memory, then Data Blocks 0..3999 cannot be written to. Data blocks 4000..7999 reside in the PCD's data memory (always RAM).
- Data Blocks share the same numbering as Texts. For example, if DB 10 is defined, then TEXT 10 is unavailable, and vice versa.
- The PCD2.M480, PCD3 and all later PCD models (NT systems) support Texts/DBs 0..8191.
- The @LEN() and @CHK() special operators can be used in Data Blocks, but only if the symbol they reference in not an external or dynamic address.

1.12 Extended Data Blocks DBX

Description

Extended Data Blocks (DBXs) are normally not created by users, they are generated by an editors code generator such as the S-Net Configurator or CAN Configurator.

The standard Data Block (DB) is an array of 32-bit Register values. This is not suitable for holding data in other formats, such as bytes, words and strings. To solve this problem, the Extended Data Block (DBX) has been introduced. DBXs are numbered 0..3999, and are stored in the read-only Text segment. They can hold any type of data.

At present, DBXs cannot be accessed by the user program, they are used only by the firmware to hold internal binary configuration data such as the Profibus configuration. It is up to the PCD's firmware to correctly interpret the contents of a DBX.

Each DBX number has a specific use which is hard-wired in the firmware. For example, DBX 3 holds the LON configuration, DBX 5 is for TCP/IP, DBX 12 is for the CAN configuration.

There is no dynamic allocation of DBX numbers. The linker does not support the DBX data type, this means that if a symbol is defined as a DBX (e.g. `MyDbx EQU DBX 100`), the symbol cannot be made Public. However, DBX symbols can be used in include files.

To make DBXs as versatile as possible, each element in the DBX is assigned a data size in bytes, see below. The default data size is 4 bytes, so that the default DBX behaves like a DB which contains 32-bit elements.

Format

```
DBX number        ;can be symbol or absolute value, 0..3999
@size:            ;data size declaration
d1,d2,d3,         ;data, each one is size bytes
@size: d4, d5     ;more data, different data size
...
EDBX              ;end of DBX
```

The DBX can store any size of data. Each data item is preceded by its size in bytes, using the form:

```
    @size:
```

where size is the data size in bytes, preceded by an ampersand `@` and separated from the data by a colon `:`.
After the size follows a list of data values, separated by commas. Each item uses the same size, until the next size declaration. Commas must be used to separate data items on the same line, they are not needed if there is only one item per line.

All values are stored in the DBX in Motorola format (MS byte first).

If the size is larger than the data value it is padded to the correct length. For text in double quotes "…", spaces are padded to the right of the text and a NUL (00) is used as the last character. For numeric data, zeros are padded to the left of the data. For example:

```
    @5: "No"      ;4E 6F 20 20 00 hex
    @5: 12H       ;00 00 00 00 12 hex
```

To remove the padding spaces from the end of a text, put one or more NUL characters <0> in the text, e.g.

```
    @5: "No<0>"   ;4E 6F 00 20 00 hex
```

If the text is the exact length-1 then the NUL is not needed because the last character of a text is always 00:

```
    @5: "1234"    ;31 32 33 34 00 hex
```

If the length of the data is larger than the current size, an error is generated. For text data, the length should include the added NUL.

Symbols can be used to declare data in a DBX. The symbol's value is converted to a binary value in the same format as a PCD instruction operand. These are for use by PCD firmware.

If the data size is `@0`, the symbol's value is stored in ASCII, and a text format can be used, e.g. `Sym1.04T`, see Using Symbols in Texts and Storing variable length text in IBs and DBXs using @0.

Media addresses

Media addresses are converted to the PCD firmware format, which is a 16-bit binary representation that contains the address and the media type code (mc). Note that Register addresses are multiplied by 4 before adding the mc (8000h). This format is for use only by the PCD firmware which processes a DBX.

IMPORTANT

Only the first media address in an expression is converted to the firmware format. Subsequent symbols in an expression are NOT converted. For example:

```
Reg1 EQU R 1
Flg2 EQU F 2
DBX 10
@2: Reg1      ;result is 8004h (registers numbers are x4)
    Flg2      ;result is 4002h
0+Reg1        ;result is 1
0+Flg2        ;result is 2
EDBX
```

Media addresses can be represented as ASCII strings instead of binary values by using the `d` or `h` text format characters (these can be used only in a DBX or IB). The address is in the same format as a PCD instruction operand, and includes the mc. For example:

```
Reg1 EQU R 1
Flg2 EQU F 2
DBX 0
@0:                  ;for variable length texts
Reg1.h, " ", Reg1.d  ;"8004 32772" (register numbers are x4)
Flg2.h, " ", Flg2.d  ;"4002 16386"
EDBX
```

NOTES

- The @LEN() and @CHK() special operators can be used in a DBX, but only if the symbol they reference in not an external or dynamic address, and the Text or DB they reference does not contain any external data.
- DBX numbers are from 0..3999, which means that they are stored in Text/DB memory only, they cannot be defined in Extension Memory at present. Therefore they are always read-only.
- DBXs are always multiples of 4 bytes in length. If the DBX is shorter then it is padded with 1, 2 or 3 bytes of NULs (0s).

Conditional Directives

Conditional directives `$IF..$NDIF` etc. and Macros can be used inside DBXs. Macros can be very useful, particularly if DBXs are coded by hand:

```
byte macro data
  @1: data
endm

word macro data
  @2: data
endm

dword macro data
  @4: data
endm
```

```
    string macro data
      @0: data
    endm

    DBX 0
    byte 255
    word 0ffffh
    dword F 100
    string "This is a string<0>"
    EDBX
```

Example
DBX example for a Profibus data structure:

```
DBX ProfibusDB    ;can be symbol or absolute value
;vfd_par
@12:"SAIA", "PCD1.M100", "", ;12-char strings, incl. NUL
@2:120H

;obj_head
@1: 0             ;spare
@1: 0             ;ram_flag
@1: 0             ;name_len
@1: 0             ;acc_pro_sup
@2: 0             ;version_ov
@4: 1234          ;add_obj_head
@2: 45            ;s_type_nbr
@4: 4567          ;add_s_type
@2: 10            ;s_obj_beg
@2: 66            ;s_obj_nbr
@4: 6789          ;add_s_obj
@2: 0             ;d_obj_beg
@2: 0             ;d_obj_nbr
@4: 0FFFFFFFFH    ;add_d_obj
@2: 0             ;d_pro_beg
@2: 0             ;d_pro_nbr
@4: 0FFFFFFFFH    ;add_d_pro

;pb_typ_tbl
@4:               ;4-byte values
0,34,67,89,97,105,;add_type1..add_type_n
120,134,149,150

;pb_type list
;Strings with @0 size are NUL terminated by
;specifically adding the NUL <0> at the end,
;See Storing variable length text in IBs and DBXs using @0.
@1:5, @1:22, @0:"This is a description<0>"
@1:5, @1:19, @0:"Another description<0>"
@1:6, @1:3,  @0:"huh<0>"
...

;pb_obj_tbl
@4                ;4-byte values
0                 ;add_obj_x+0
28                ;add_obj_x+1
```

```
56                  ;add_obj_x+2

;pb_object list
;pb_object_1
@1: 0               ;spare
@1: 4               ;obj_code
@2: 1               ;type_ind
@1: 1               ;obj_len
@1: 8               ;nb_element
@1: 0AFH            ;password
@1: 1               ;acc_group
@2: 0034H           ;acc_right
@4: 000083C0H       ;add_media
@1: 0               ;name
@1: 01              ;media_len
@1: 0               ;spare
@2: 8               ;count
@4: 1234            ;w_ind_add
@4: 1235            ;r_ind_add

;pb_object_2, as above but condensed without comments
@1:0,4,@2:1,@1:1,8,0AFH,1,@2:0034H,@4:000083C0H
@1:0,01,0,@2:8,@4:1234,1235

;other pb_objects can follow
...
EDBX
```

1.13 Information Blocks IB

Description
Information Blocks (IBs) are not normally coded by users, they are generated by an editor's code generator.

IBs are similar to DBXs, and are declared in the same way, except that `IB..EIB` is used instead of `DBX..EDBX`. Up to 4000 IBs can be defined, numbered 0..3999.

The only two differences between DBXs and IBs are that IBs are not downloaded into the PCD when the .PCD file is downloaded, and the default data size for IBs is 0 (@0 for variable length texts) instead of 4 for DBXs.

IBs are stored in a hidden segment in the .PCD file. This segment can be read from the .PCD file only by specially written PC software, it cannot be accessed by the PCD itself.

IBs are always multiples of 4 bytes in length. If the IB data is shorter then it is padded with 1, 2 or 3 bytes of NULs (0s).

The linker does not support the IB data type, this means that if a symbol is defined as an IB (e.g. MyIb EQU IB 100), the symbol cannot be made public. However, IB symbols can be used in include files

Format
See DBX.

Example
```
IB 0
```

```
   "This is some text with a NUL terminator<0>"
   "This is more text with a carriage return terminator<CR>"
   Reg1.004T       ;"R0001" see Using symbols in texts
   Flg2.h          ;"4002"  ''
EIB
```

See also Storing variable length text in IBs and DBXs using @0.

1.14 Comments

Description
Comments can appear anywhere in the source file (except after some directives such as $TITLE, $ERROR, $REPORT). Comments begin with a semicolon character ; and can be any length, but for symbol comments only the first 80 characters are used.

The comments appearing after symbol declarations (PEQU, EQU and DOC etc) are stored in the symbol table in the PCD file.

All characters after the ; are ignored until the start of the next line.

Comments can contain any characters.

Format
```
;comment
```

Tip: Comments in Macro definitions which are preceded by two semicolons ;; are removed when the macro is expanded, this saves a bit of memory and makes the object files slightly smaller. (On an old PC with 640KB RAM this feature was useful, but if you have a new PC with 64MB RAM, it is not really significant anymore.)

See also
$SKIP..$ENDSKIP

1.15 Strings, STR and @STR( )

Description
A *string* is a bit like a macro parameter, except the string name can be used anywhere in the IL program, not just inside a macro. The string is replaced with the actual text during the pre-processing pass of the assembler. Some new FBox Adjust parameters are *strings* - not symbols or values, but simply some textual information. A string is not a Text (as in Texts and Data Blocks), but it can be used to define Texts, see the example below.

Defining a string
Strings can have names and can be defined using DEF, EQU, LDEF, GEQU or GDEF, and the data type STR. The string's text is enclosed in double quotes "...".

```
  string_name DEF|EQU|LDEF|GEQU|GDEF STR "string"
```

The quotes are removed when the string is referenced using the @STR( ) operator, which is described below.

String symbol names are kept in a separate symbol table - the string table, so their names will not clash with normal symbol names. String symbol names are valid from the point of definition to the end of the file. Forward references are not allowed, strings must be defined before they can be used.

String definition - the quotes are removed when it is referenced using @STR( )
```
  MyString DEF STR "Strings are cool"
```

If you want to keep the quotes, use two pairs of quotes as in this example - @STR( ) removes only the outer quotes.
```
  MyString DEF STR ""Keep the quotes""   ;@STR() removes only the outer quotes
```

Referencing a string

Strings can only by referenced using the special operator @STR( ), or in the $IFDEF or $IFNDEF conditional directives. The @STR( ) operator is replaced by the string itself, without the quotes (unless a double pair of quotes was used when the string was defined). See the examples at the end of this page.
```
  MyString DEF STR ""Keep the quotes"" ;@STR() removes the outer quotes
  @STR(MyString)                       ;evaluates to "Keep the quotes"
```

Note: No space is allowed between @STR and the (, you must use `@STR(`.

@STR( ) can be used directly in these directives for text output: $REPORT, $WARNING, $WRFILE etc. It is not necessary to enclose this operator in @...@ characters to enable it to be evaluated ***unless it is a macro parameter (see below)***, using `@STR()` alone is the same as using: `@&$STR()@`. @STR( ) operators are resolved after macros have been expanded, and before the code is assembled. This allows string names to be passed as macro parameters, and allows @STR( ) to reference a string defined outside the macro, or in a caller macro (see GDEF). For example:
```
  String1 DEF STR "Strings"
  String2 DEF STR "confusing"
  $WRFILE "Test.txt" @STR(String1) are very @STR(String2)!
```
The text written to file Test.txt is:
```
  Strings are very confusing!
```

String names can be derived from other string names, but you must still use STR otherwise S-Asm will not look in the string table:
```
  DerivedString EQU STR MyString
```

String names can be used in $IFDEF or $IFNDEF to see if the string name has been defined or not, you must also use STR to indicate it's a string, otherwise S-Asm will look in the symbol table instead of the string table:
```
  $IFDEF STR MyString
  ...
  $ENDIF
```

Tip: The results of the string replacement can be seen in the Listing file. Examine the listings while learning to use @STR( ).

Using strings as macro parameters

It is usually best to pass a string name rather than the string itself as a macro parameter. But you can also pass an actual string "hello" or an @STR( ) expression, but strings with two pairs of quotes `""...""` are not supported as direct macro parameters. Note also that actual strings cannot contain a `"` character.
Special handling is needed for macro parameters in the $WRFILE, $REPORT, $ERROR, $WARNING and $REPORT directives because these directives assume that what follows is text to be output as it is, without any processing. Using a parameter name on its own will not work, so you must indicate it is to be processed by enclosing it in `@&...@`.
@STR( ) is not replaced in a MACRO definition, it is replaced only when the macro is expanded.
@STR( ) is replaced after the macro parameters are replaced.
For example, here 'param' is the macro parameter which is passed as @STR("...") :
```
  MyMacro MACRO param1
```

```
    $REPORT This is the string: @&param1@
  ENDM

  MyMacro(@STR("Goodbye, world."))
```

`@STRLEN(`*`string_name`*`)` can be used to get the length of a string. This returns -1 if the *`string_name`* is empty, and so can be used to detect empty macro parameters.

`$IFTYPE` can be used to determine if a symbol name is a string name:

```
  $IFTYPE string_name = STR
  ...
  $ENDIF
```

Referencing macro parameters inside a string

From PG5 V2.1.300 (S.SYS.SasmVers >= 21300), macro parameters can be referenced from inside a atring by using `"@&`*`param`*`@"`. Without this, the macro parameter '*`param`*' is not replaced. For example:

```
  DemoMacro MACRO param0, param1
  String1 EQU STR "Macro parameters are: '@&param0@' and '@&param1@'"
  $RPEORT @STR(String1)
  ENDM
```

Defining a TEXT from a string

Texts must be enclosed in quotes "...", so you can either declare the string with two pairs of quotes (the outer pair are removed) and not enter the quotes in the TEXT, or you can declare the string with a single pair of quotes (which are removed) and put the quotes into the TEXT. The example below shows both methods.

```
  String0 EQU STR ""Part 1""  ;string with "quotes"
  String1 EQU STR "Part 2"    ;string without quotes
  TEXT 100 @STR(String0), "@STR(String1)"
```

The resulting Text is:

```
  TEXT 100 "Part1", "Part 2"
```

Notes

- String names are not affected by $GROUP, but they can have a dot in the string name, for example:

```
  Group0.String0 EQU STR "I'm a groupie"
  $GROUP Group0
    String1 EQU STR "I'm not in a group"
  $ENDGROUP
```

- Strings cannot be Public, the string table is local to each file.
- There is no cross-reference list for strings.
- String names are not shown in Project Manager's 'Data List' view.

Notes for library developers

Strings have been introduced in PG5 V2.0, S-Asm version 2.0.150. They are not supported by earlier versions. This means that any libraries or FBoxes which use strings must check the S-Asm version number using the pre-defined symbol S.SYS.SASMVERS:

```
  $IF S.SYS.SASMVERS < 2000150
    $FATAL MyLibrary requires PG5 version 2.0.150 or later
  $ENDIF
```

Tip: Library developers can also use an entry in the Library Information File (.saialin) to prevent installation of the library if the PG5 doesn't support it, see section `[AppVersion]`.

Pre-defined System Strings

These strings are pre-defined for use in IL programs:

`S.STR.PG5Registere`	Registered user name of Saia PG5® package
`S.STR.PCUserName`	Name of the user who is logged onto the PC.
`S.STR.PG5Version`	Version number of PG5, e.g. "2.0.100"
`S.STR.ProjectName`	The name of the project
`S.STR.DeviceName`	The name of the device (program name)
`S.STR.PcdType`	The PCD type, e.g. "PCD3.M5540"
`S.STR.ProgramVers`	Program version for the Device Properties dialog box, e.g. "1.0"
`S.STR.ProgramID`	The unique program ID
`S.STR.FileName`	The name of the source file.
`S.STR.AppLanguage`	Language selected for applications, "en", "fr", "de", "it" etc.
`S.STR.LibLanguage`	Language selected for libraries, "en", "fr", "de", "it" etc.

See also

@ATTR( ) References a symbol's attribute string.
@STRLEN( ) Gets the length of a string

Examples

```
;Defining a string
;use EQU, DEF, LDEF, LEQU, GDEF or GEQU
;use new STR type
;cannot be PUBL
;string names are stored in a separate String Table
StringName0 EQU STR "This is a string"

;String names can also be derived from an existing string name
StringName1 EQU STR StringName0

;String names are not affected by $GROUP
;but you can give them a group name like this
Group0.String0 EQU STR "String theory is history"

;To reference a string, use the @STR( ) operator
;the entire operator is replaced by the string with the quotes removed
;Strings are replaced by the preprocessor after macros
;have been expanded but before the line is assembled
TEXT 100 "@STR(StringName0)"

;To keep the quotes, define a string with two pairs of quotes
;the outer quotes are removed
StringName2 EQU STR ""String with quotes""
TEXT 102 "@STR(StringName0), ", @STR(StringName2)

;@STR( ) will also work for absolute strings
;so a string name or the string itself (in quotes) can be used
;the outer pair of quotes are removed
TEXT 101 "@STR("String without quotes")", @STR(""String with quotes"")

;String concatenation
;simply use more than one @STR() on the line
;but don't forget the quotes around the new string!
BigString EQU STR "@STR(StringName0)@STR(StringName1)"

;String names or absolute strings can be passed as macro parameters
```

```
StringMacro MACRO textNum, Param1, Param2
TEXT TextNum "@STR(Param1)", @STR(Param2)
ENDM

StringMacro(200, "String without quotes", ""String with quotes"")
StringMacro(201, StringName0, StringName2)

;Strings can be used in $IFDEF
;use STR so S-Asm knows it must look in the string table
$IFDEF STR StringName0
  $REPORT StringName0 is defined as "@STR(StringName0)"
$ENDIF

;In fact, strings can be used anywhere
String4 EQU STR "R 123"
    ...
    FB  0
    LD  @STR(String4)
        666
    EFB

;Using strings in $WRFILE (we use $REPORT for the demo)
;there is no need for @...@
$REPORT "Test1.txt" @STR(StringName0), @STR(StringName2)

;Only the new version of PG5 can be used, this should be checked
;in the IL code
$IF S.SYS.SASMVERS < 2000150
  $FATAL MyLibrary requires PG5 2.0.150 or later
$ENDIF

;Can also use [AppVersions] in .saialin when installing the library


;Symbol Attributes
;defined with $ATTR and referenced with @ATTR()
;@ATTR() can be used in exactly the same way as @STR()

$ATTR MaxTemp=120
$ATTR MinTemp=-20
SymbolWithAttribute EQU R 100  ;symbol's comment

COB 0
    0
LD  R 100
    @ATTR(SymbolWithAttribute, MaxTemp)
LD  R 101
    @ATTR(SymbolWithAttribute, MinTemp)
ECOB

;There are also pre-defined attribute names
; Type | Value | Expression | Comment | Symbol
; attribute names are not case sensitive
Symbol110 EQU R 100+10 ;Comment for Symbol110

$REPORT @ATTR(Symbol110, Type) = R
```

```
$REPORT @ATTR(Symbol110, Value) = 110
$REPORT @ATTR(Symbol110, Expression) = 100+10
$REPORT @ATTR(Symbol110, Comment) = Comment from Symbol110

;Strings can be declared from symbol attributes
;but don't forget the quotes!
AttributeString EQU STR "@ATTR(SymbolWithAttribute, Comment)"

;Strings can also be created from a symbol name
;using the pre-defined attribute: Symbol
StrSymbol110 EQU STR "@ATTR(Symbol110, Symbol)"
    ...
    INC @STR(StrSymbol110)  ;same as: INC Symbol110


;Strings can be passed from one macro call to another

macro2 MACRO param1, param2
  $REPORT @&param1@ and @&param2@
endm

macro1 MACRO param1, param2
  macro2(param1, param2)
endm

macro1( @STR("hello"), @STR("goodbye") )


;That's all folks
```

1.16 Reserved Words

The following words are reserved, and cannot be used as symbol names.

Note: Some of these reserved words cannot be used because they are single characters, and symbols must be more than one character long.

Assembler declarators

```
PUBL  EXTN  DOC  EQU  DEF  MACRO  LEQU  LDEF  GEQU  GDEF  PEQU
TEQU  OEQU  XEQU  ENDM  EXITM IN  INOUT
```

Medium control codes and data types

```
I  O  I|O  F  T  C  T|C  R  K  M  X  DB  TEXT  ST  TR  PB  FB
SB  COB  XOB  IST  S  SEMA  EDB  DBX  EDBX  IB  EIB  VOL  RAM
BOOL  INT  UINT  FLOAT  IEEE  DOUBLE  STR
```

MOV instruction special codes

```
Q  B  W  L  D
```

Condition codes

```
H  L  P  Z  N  E
```

New condition codes (reserved for future use)

```
GT  LE  GE  LT  EQ  NE  VC  VS  GTU  LEU  GEU  LTU  CC  CS
```

Instruction mnemonics

To see the full list of all reserved words, you can look in the IL Editor's syntax checking files, called SeditSyntax*n*.dat. These are text files which contain the full list of reserved words and mnemonics. They can be found in the "Local Directory", see Project Manager's "Tools / Options / Directories" for the Local Directory path.

Pre-defined symbols
Symbols which are defined by the system.

Symbols which begin with an underscore
Symbols beginning with an underscore are reserved for use as internal symbols or in libraries.

1.17 Pre-defined Symbols

Symbols Generated by S-Asm
The assembler defines some useful symbols internally. These can be referenced by the user program as normal symbols.

`_ArraySize_.<symbol>`	For every symbol which is defined as an array, another symbol is automatically generated which is assigned the size of array. The symbol name is prefixed with `_ArraySize_`.
	For example:
	`Symbol EQU R [10]`
	This generates the internal symbol:
	`_ArraySize_.Symbol EQU 10`
	If the array symbol is Public, yhen the _ArraySize_ symbol will also be public.
	
	Tip: You can see these in Project Manager's "Data List" view by checking the "Internal Symbols" option in the Data List Filter.

`_BLOCKNUM_`	The number of the current block, or -1 (`0FFFFFFFFH`) if outside a block.
	`PB   123`
	`LD   R 0          ;same as LD R 0`
	`     _BLOCKNUM_   ;        123`
	`EPB`

`_BLOCKTYP_`	The type of the current block: -1 = outside block (`0FFFFFFFFH`) 0 = COB, 1 = XOB, 2 = PB, 3 = FB, 4 = IST, 5 = ST, 6 = TR
`__SASMVERS__`	**NOTE:** This has been superseded by `S.SYS.SASMVERS`, see below. The version number of the assembler/linker, as a 5-digit decimal integer. E.g. V1.4.040 = 14040, $1.4.041 = 14041.
`__PGVERS__`	The PG programming package version number, 5 = PG5.
`__PGBUILD__`	The PG programming package build number (software version), e.g. V2.0.100 = 20100.
`__PCD_UID__`	'Unique program identifier' data block number. This is used by the 'download changed blocks' feature to identify the target PCD.

System Symbols
System Symbols all begin with the group name "S". There are many system symbols, and new

system symbols are being added all the time.

Below are just some of the system symbols and system symbol groups, not all symbols are listed because new System Symbols are being added for every PG5 release.
Tip: To see all the available System Symbols, open the Data List view from Project Manager. Each system symbol has a description comment.

`S.CPU.PcdType`	Defines the PCD device type configured in the Device Configurator. It is assigned to a symbol for the PCD type, e.g. `S.CPU.PcdType EQU __PCD3_M5540__`. This symbol can be used in $IF to generate code for a specific PCD type.
`S.SYS.SasmVers`	System symbol which has the value if the version of the assembler/ linker in the new 9-digit format "aaabbbccc". 'aaa' is the major version, 'bbb' is the minor version and 'ccc' is the build number. e.g. V2.0.100 would be 002000100. This can be used in $IF statements for S-Asm feature checks. Use this version number instead of the old `__SASMVERS__`.
`S.CPU.xxx`	System symbols created by the Device Configurator for each PCD device.
`S.PRJ.xxx`	System symbols created by Project Manager for the entire project. See tip above.
`S.ODM.xxx`	Open Data Mode system symbols created by the Device Configurator.
`S.RIO.xxx`	Diagnostic and base address symbols created by the RIO Network Configurator.

Pre-defined System Strings
Many of the Program Information items are available as *strings*, see Strings, STR and @STR( ).
Examine the _devicename.inc file to see what strings are available.

```
;System Strings
S.STR.PG5Licensee EQU STR "Matt Harvey"
S.STR.PG5DeveloperID EQU STR "007"
S.STR.PG5Version EQU STR "$2.1.46.1 Patch 1"
S.STR.ProjectName EQU STR "Project1"
S.STR.DeviceName EQU STR "Device1"
S.STR.PcdType EQU STR "PCD3.M5540"
S.STR.ProgramVersion EQU STR "1.0"
S.STR.ProgramID EQU STR "A1DA8BCED93BAEF0"
S.STR.AppLanguage EQU STR "en"
S.STR.LibLanguage EQU STR "en"
```

Symbol Prefixes
These internal symbol prefixes are generated by the assembler:

`__LEN__xxx`	Generated by the @LEN(...) special operator, xxx is generated from @LEN's parameters.
`__CHK__xxx`	Generated by the @CHK(...) special operator, xxx is generated from @CHK's

	parameters.
`_G_xxx`	Prefix used for GEQU and GDEF symbols. xxx is the symbol name.
`__ADS__xxx`	Generated by the @ADDS(...) special operator, xxx is created from the @ADDS() parameter.
`__abs__xxx`	Dummy symbol for absolute address. xxx is the type and value, e.g. __abs__F_123, __abs__COB_0, __abs_TEXT_45.

Library Version Symbols
These symbols allow access to a library version as a symbol. The symbol can be used to determine which versions of a library were used in the program.
They can also be used with S-Asm's conditional assembly directives $IFxxx. to generate code for a specific library version.
The new `__LIBVERS__` symbol group contains the version numbers of all selected libraries.
The symbols are created in include files which make s the symbols available to all files in the program. The symbols can be seen in Symbol Editor;s "All Symbols" view, or in Project Manager's "Data List" view.

For example, these are the declarations in the include file "_device.inc":

```
;Library version symbols
$GROUP __LIBVERS__
_SAIA_D3T76XH100 EQU 001001000   ;D3T76xH100 RIO, V1.01.0
_SAIA_D3T76XH110 EQU 001000000   ;D3T76xH110 RIO, V1.0.0
_SAIA_D3T76XH150 EQU 001001000   ;D3T76xH150 RIO, V1.01.0
_SAIA_D2H110 EQU 002002000 ;PCD2.H110 Counting Module, V2.02.0
_SAIA_D2H150 EQU 002002000 ;PCD2.H150 SSI Encoder Module, V2.02.0
_SAIA_D2H210 EQU 002002000 ;PCD2.H210 Stepper Motor Positioning, V2.02.0
_SAIA_D2H310 EQU 002001000 ;PCD2.H310 Servo Motor Control Module, V2.01.0
_SAIA_D2H320 EQU 002044000 ;PCD2.H320 Motion Control Module, V2.44.0
_SAIA_W745 EQU 002002000   ;PCD2.W745 Thermocouple Module, V2.02.0
_SAIA_W3X5 EQU 002010000   ;PCD2/3.W3x5 Analogue Input Module, V2.10.0
_SAIA_W6X5 EQU 002010000   ;PCD2/3.W6x5 Analogue Output Module, V2.10.0
_SAIA_W800 EQU 002002000   ;PCD3.W800 Analogue Output Module, V2.02.0
_SAIA_SFUPBASE EQU 002006100     ;S-Fup Base Library, B2.6.100
$ENDGROUP __LIBVERS__
```

1.18 Initializing Data

There are several occasions when data needs to be initialized. The first time a program has ever been run (first-time initialization) , or every time a program is started (start-up initialization, on restart or power-up), or every time a block is run (run-time initialization).

Start-up Initialization
Start-up initialization is done on every power up or restart of the PCD. All volatile data values are set to zero, but all non-volatile data retains the values it had before the power off or restart. Some of this data will need to be initialized with the correct values before it is used. This is normally done by code in XOB 16.

Flags can be partitioned into volatile and non-volatile sections, using Project Managers Build Options and the DEFVM instruction. All Registers are always non-volatile.

At present there is no built-in support for startup initialization, so this must be coded in the start-up XOB 16 or between the $INIT..$ENDINIT directives which defines code that is inserted into XOB 16.

Alternatively, you could use code like that in the **Tip** example below to initialize the data of a block the first time the block is called, but using a volatile flag for each block which is always zero the first time the block is called. The data is initialized if the Flags is 0, and the Flag is set to 1 once it has been initialized.

NOTE
In the future, support for startup initialization may provided by the PCD firmware. If it is, then this syntax may be used for startup initialization:
```
Symbol EQU R = 100
```

Tip: Initialization can easily be done by IL code. Values which need to be initialized only once could be initialized whenever a single volatile Flag is found to be zero, so the initialization code is run only the first time the block is called after start-up. A non-volatile flag could be used for "first-time initialization data".

```
    FB      MyBlock

    ;Initialize static data on first call
    MyBlock.InitFlag EQU F VOL  ;volatile flag, always 0 on start-up
    STH     MyBlock.InitFlag    ;has data been initialized yet?
    JR      H AlreadyInit       ;yes, skip initialization
    ACC     H                   ;no, set flag and do initialization
    SET     MyBlockInitFlag
    ...                         ;do start-up initialization here

AlreadyInit:
    ...                         ;rest of program
    EFB
```

First-time Initialization

NOTE
Do not use First-time Initialization if the data will be initialized in start-up XOB 16. This is start-up initialization, and first-time initialization is not needed.

Normally, data is initialized every time the program runs (start-up initialization), usually by code in the start-up XOB 16 or by code in each block. Volatile data is also set to zero on every restart or power up of the PCD, so that doesn't need to be initialized if its starting value will be zero. But for non-volatile data, whose values are not set to zero on power-up or restart, you may want to initialize these only once and keep the values for the lifetime of the PCD or process (or until the battery runs out).

This could be useful for a permanent counter which might contain, for example, the total number of hours that a machine has been run, a light has been on, or a door has been open. Such values need to be initialized only once. This can be done using the **first-time initialization data** feature. The initialization values are defined in the EQU or PEQU declaration of the Register, Timer, Counter or Flag symbol, using "`:=`":
```
    MasterCounter EQU C 100 := 0
```

Registers, Counters and Flags are non-volatile (for Flags see DEFVM instruction). This means that they are not initialized and can contain any unknown values when the program runs for the first time. So they need to be initialized by code in the user program before they are used. This can be done every time the program starts, or can be done only once when the program is downloaded.

Project Manager's Download Program dialog box has a checkbox which allows these values to be downloaded separately. They can be downloaded at the same time as the program, or separately by using the "First-time initialization data only" checkbox.

First-time initialization values are stored in a record in the PCD file (IB or DBX).

The initialization value can be any expression, even using other symbols, but the value cannot be External.

Examples of first-time initialization data

```
Symbol1  EQU R 100 := 0             ;R 100 initialized to 0 on download
Symbol2  EQU R     := A_Symbol+4    ;expressions can be used
Array1   EQU R [5] := 1,2,3,4,5     ;initializes a 5-register array
FlagSym  EQU F     := 1             ;must be 01 or 1 for Flags
```

Tip: This can also be done using a single *non-volatile* Flag, as in the example for startup initialization above.

Run-time Initialization

Work data needs to be initialized every time a block runs. If using temporary data, it is always initialized to zero. But if using normal Registers and Flags etc, these can contain any values when the block is called, so they must be initialized before they can be used. This is normally done by code in the user program which initializes the data just before it is used. So this doesn't normally cause any problems when downloading changed blocks in Run or in Stop.

1.19 Dynamic Address Allocation

In many cases the actual address of data such as a Register or Flag is not important. For example, it makes no difference to the program which Register is used to hold a temporary value, providing it is not used by another part of the program while the data it holds is still needed. "Dynamic address allocation" means that you do not need to assign unique address to every symbol, addresses can be automatically assigned by S-Asm at build time. This is also known as "Automatic address allocation".

Once an address has been assigned to a symbol, it will not change for every build. It will remain the same until the user resets the dynamic address allocator using Project Manager's Device / Clean Files command. Dynamic addresses are store in the 'Symbol Information' files, see description below.

Data types which can be dynamically assigned are:
Registers, Timers, Counters, Volatile Flags, Nonvolatile Flags, Texts, RAM Texts, DBs and RAM DBs.

Code block numbers (except XOBs) can also be dynamically allocated.

The range of addresses used for dynamic address allocation for each data type is defined in SPM's Build Options for each device. The range for code block numbers always starts from 0.

Once the ranges for dynamic address allocation have been defined, symbols can be assigned with EQUate statements, or in the Symbol Editor, leaving out the address - only the type is required. The linker will assign absolute values to these symbols from within the dynamic address range. For example:

```
    WorkReg1    EQU R       ;Reserve single workspace elements
    WorkReg2    EQU R
    TempFlag1   EQU F
    TempCounter EQU C
```

To reserve an array of addresses, an optional array size can be given, enclosed in square brackets:

```
    TenRegs     EQU R [10]  ;Reserve 10 registers
    FiveFlags   EQU F [5]   ;Reserve 5 flags
```

This is useful if an offset is used when referencing the symbols, and saves having to assign a symbol for every element used:

```
    ADD   TenRegs
          TenRegs+1
          TenRegs+2
    STH   FiveFlags+4       ;References the last flag
```

Dynamically assigned addresses can be used in the same way as symbols with absolute addresses. They can also be made Public.

In the listing file, automatically assigned symbols are shown with an "A" in the external (E) column, and "AUTO" in the scope column of the cross-reference listing. They are also shown with AUTO in Project Manager's "Data List" view.

Volatile Flags

To allocate Flag addresses from the "Dynamic Volatile Flags" range, you must use the F VOL data type, otherwise the Flag address will be allocated form the "Dynamic Nonvolatile Flags" range. For example:

```
FlagA    EQU F VOL    ;volatile flag
FlagB    EQU F        ;nonvolatile flag
```

Texts and DBs in Extension Memory (RAM)

To allocate Text or DB addresses from Extension Memory (writable RAM), you must use the TEXT RAM or DB RAM data type, otherwise the address will be allocated from the normal Text/DB memory (which may be read-only). For example:

```
WritableText  EQU TEXT RAM  ;Text in Extension Memory
ReadOnlyText  EQU TEXT      ;Text in Text/DB Memory
```

WARNING

It is not guaranteed that dynamic addresses will never change, so please do not rely on these addresses being fixed. It is recommended that dynamic addresses are never used if the address is accessed by an external system or programming tool. Always use a fixed absolute address (e.g. R 123) which you know will never change.

For example, a supervision system should never access a dynamically addressed symbol by an absolute address, because if the address changes then the supervision system will be accessing the wrong data without knowing it!

Dynamic addresses are retained mainly to support the "download changed blocks" and "download in run" features, so that we don't have to download all the blocks as we would if all the dynamic addresses were changed by every build. This would change every block that uses them and the whole program would always have to be downloaded.

Dynamic addresses could be re-assigned without notice if a fatal build error occurs, or if the user restores or copies a program without the "symbol information files". Certain fatal errors could also cause the symbol information files to be deleted or not updated, and the "Clean Files" command could be used at any time. This can happen without notifying the user - he will not know that the addresses have changed!

Tip: For Public symbols, the actual address can be read by an external system or programming tool by looking in a text file called "_Global.sy5". This file is updated by every build, and contains a list of all Public symbols, their types and actual addresses. After every (successful) build, this file could be read or imported to get the actual addresses. It is also possible to read the symbol table in the PCD file by using functions in Sasm52.dll from a C, C++ or C# program, see document "PG5 V2 Sasm

API.doc" .

Tip: You can now use temporary data (see TEQU) for R and F values which are only needed when the block is running.

Symbol Information Files
Once S-Asm has assigned an address to a symbol, the address will usually stay the same even if the user program is changed. This is done by storing a list of symbols and their assigned addresses in 'symbol information files'. These files are in the 'Sym' subdirectory, and have the file type '.si'.
If the dynamic address range is changed from the Project Manager's 'Build Options' dialog box, then all addresses will be re-assigned.
To for all the addresses to be re-assigned, use Project Manager's 'Device / Advanced > / Clean Files...' command. After many builds, the dynamic addresses may become fragmented because many symbols have been added or removed. 'Clean Files' deletes the symbol information files and causes all the addresses to be consecutively assigned on the next build.

1.20 Storing Variable Length Text in IBs and DBXs Using @0

An IB or DBX can store text strings.

The size of 0 (@0) is used to introduce text data. The size of the data is defined by the length of the string. Text data is entered in the same way as Texts. Characters can also be entered in decimal or hex by enclosing the value in angle brackets (e.g. `"<10><CR><0Dh>"`), and the text can contain symbols and formatting information, see Using Symbols in Texts.

Note that the NUL (0) terminator is not automatically added to the end of the text, if a NUL is required it must be explicitly added by putting `<0>` at the end of the text, e.g.

```
"This is a NUL terminated text<0>"

;DBX containing variable-length texts
DBX 100
@0: "This is line 1, NUL terminated <CR><LF><NUL>"
@0: "This is another line 2 <CR><LF><NUL>"
    ;Only the first @0 is needed, the last @size is retained
    ;until another @size is found
    "This is line 3<13><10><0>"
EDBX
```

1.21 XOB List

Each Exception Organization Block (XOB) has a specific function.

XOB	Description	Priority
0	Power down	4
1	Power down in extension rack	1
2	Low battery	1
3	Task/Temp Data overflow	3
4	Parity error on main bus (PCD6 only)	1
5	No response from I/O module	1
6	External error	1
7	System overload	3

8	Invalid opcode	4
9	Too many active tasks (GRAFTEC)	1
10	PB / FB nesting depth overflow	1
11	COB supervision time exceeded	2
12	Index register overflow	1
13	Error flag set	3
14	Cyclic XOB	2
15	Cyclic XOB	2
16	Cold Start	2
17	S-Bus XOB Interrupt Request	2
18	S-Bus XOB Interrupt Request	2
19	S-Bus XOB Interrupt Request	2
20	Interrupt input IN0 / Interrupt input INB0 **(1)**	2
21	Interrupt input IN1	2
22	Interrupt input IN2	2
23	Interrupt input IN3	2
24		
25	Time Cyclic Alarm / Interrupt input INB1 **(1)**	2
26	Time Cyclic Alarm	2
27	Time Cyclic Alarm	2
28	Time Cyclic Alarm	2
29	Time Cyclic Alarm	2
30	RIO connection master slave	1

(1) For PCD1 and PCD2.M1xx, XOBs 20 and 21 are Interrupt inputs INB0 and INB1 respectively.

Exception Priorities
There are 4 priority levels for XOBs. Note that XOB priorities are slightly different for the older PCDs.

Level 4 exceptions (highest)
Priority level 4 is the highest priority, only XOBs 0 and 8 can interrupt execution of another XOB.

Level 2 and 3 exceptions
If a level 2 or 3 exception occurs during execution of a lower priority XOB, then it will be run directly after the end of the current level XOB.

Level 1 exceptions (lowest)
Any level 1 exception which occurs during another exception will never be handled.

Level 4 Exceptions
Priority level 4 is the highest priority, only XOB 0 and 8 can interrupt execution of another XOB.

XOB 0 Power Down
There can be up to 10ms between the call of XOB 0 and the final loss of power to the PCD to give the user time to perform some urgent saves of values.
If the XOB 0 is programmed then the message "XOB 0 START EXEC" is written into the History List at the start of the XOB and "XOB 0 EXECUTED" upon completion of the XOB, this indicates to the user that the XOB completed before power was lost.

If the XOB is not programmed then a restart cold is immediately performed upon detection of the power down. If the XOB is programmed then a restart cold is performed upon completion of the XOB if there is still power.

XOB 8 Invalid Opcode
XOB 8 is called when the firmware detects an invalid instruction in the user program.

Level 3 Exceptions
If a level 2 or 3 exception occurs during execution of a lower priority XOB, then it will be run directly after after the current level XOB.
XOB 20/25/11 have been given a higher priority level so that if the XOB is provoked during execution of a lower or equal priority then it will be executed directly after completion of the current XOB.

XOB 3 Temp/Task Data Overflow

XOB 7 System Overload
The queuing mechanism for the level 3 XOB's has overloaded.

XOB 13 Error Flag
XOB 13 is always called when the Error flag is set by an invalid instruction, calculation, data transfer or communications error.

Level 2 Exceptions

XOB 11 COB Supervision Time exceeded
If the second line of the COB instruction indicates a monitoring time (in 1/100 seconds) and if COB processing time exceeds this defined duration, XOB 11 is called.
COB processing time is the time which can elapse between the COB and ECOB instructions.

XOB 14 Cyclic XOB
XOB 15
XOB 14 and 15 are called periodically with a frequency ranging from 5 ms to 1000s. This frequency can be set using the SYSWR instruction.

XOB 16 Cold Start
XOB 16 is the start-up XOB (Cold Start XOB), and is executed when the PCD is switched on, or is given a cold restart. XOB 16 can initialise any elements before the program begins.
If during the execution of the XOB 16 an error occurs, the XOB 13 is not called.

XOB 17 S-Bus XOB Interrupt Request
XOB 18
XOB 19
These three XOBs are started by a message on the S-Bus network; it is also possible to start them with the SYSWR instruction.

XOB 20..25 Interrupt Inputs IN0..IN3 (NT systems)
Executed on a rising edge of interrupt inputs IN0 to IN3.

XOB 20 and 25 Interrupt Inputs INB0 and INB1 (PCD1 and PCD2M1xx only)
These XOBs are called when interrupt input INB1 (resp INB2) of the PCD1/2 has detected a rising edge (see PCD1/2 hardware manual for further details).

Level 1 Exceptions

Lowest priority. Any level 1 exception which occurs during another exception will never be treated.

XOB 1 Power down in extension rack
The voltage monitor in the supply module of an extension rack (PCD 2 or PCD6) detected an excessive drop in voltage.
In this case all Outputs of the extension rack are set low within 2ms and XOB 1 is invoked.
If Outputs from this "dead" extension rack continue to be handled (set, reset or polled) by the user program in any CPU, XOB 4 and/or XOB 5 are also invoked. (Only PCD4).
XOB 1 will be called once up to 250 ms after detection of the error.
SYSWR can be used to change the behavior of XOBs 1 and 2.

XOB 2 Battery failure or low battery
The battery is low, has failed or is missing.
Information in non-volatile Flags, Registers or the user program in RAM as well as the hardware clock may be altered.
XOB 2 is called by CPU 0 every 250 ms in the event of this error.
SYSWR can be used to change the behavior of XOBs 1 and 2.

XOB 4 Parity Failure
XOB 4 can only be invoked with PCD having extension racks (PCD6 only).
The monitor circuit of the address bus has noticed a parity error. This can either arise from a faulty extension cable, a defective extension rack or from a bus extension module, or else it is simply because the extension rack addressed is not present.

XOB 5 No response from I/O module (I/O Quit Failure)
The PCD's Input and Output modules return a signal to the CPU which has addressed them. If this signal is not returned, then XOB 5 is called.
Generally, this occurs if the module is not present, but it can also happen in the case of faulty address decoding on the module.
This mechanism is not implemented on the PCD1 and 2.

XOB 6 External error
Not used. (Foreseen for intelligent modules of the PCD6)

XOB 9 Too many Graftec tasks
More than 32 Graftec branches were simultaneously activated in a Sequential Block (SB).

XOB 10 More than 7 nested PB/FB calls
PBs and FBs can be nested to a depth of 7 levels. An additional call (calling the 8th level) results in XOB 10 executing.
The 8th level call is not executed.

XOB 12 Index Register overflow
If a program contains an indexed element which falls outside its address range (0 to 8191), then XOB 12 is called.

XOB 30 RIO connection master / slaves
After every message sent from the master to a slave, the connection is tested. If the test is not answered positively by the slave, the master CPU calls XOB 30.
This is essentially the case when, online, a station is removed from the network or closed down.

1.22 IL Programming Tips

Here is a list of tips, tricks and and frequently-asked-questions for IL programmers.

Avoid common programming mistakes

1) Always use symbol names, and never mix symbols and absolute addresses for the same data

Never write code like this:

```
Symbol EQU R 123
...
INC   Symbol
...
INC   R 123
...
```

This may seem obvious, but we have seen a lot of code like this.

2) Offsets to symbols which are not arrays

If addresses must be consecutive, define an array and use offsets from the array symbol:

Good:

```
ArrayBase EQU R 100[3]
Sym0 EQU ArrayBase+0
Sym1 EQU ArrayBase+1
Sym2 EQU ArrayBase+2
```

Not good:

```
Symbol EQU R 100
AnotherSymbol EQU R 101
...
INC   Symbol
INC   Symbol+1  ;this increments AnotherSymbol, probably a bug
INC   Symbol+2
...
```

Tip: Use the Build Option "Warn on offset to symbol which is not an array" to check for these during development.

This is often attempted when an array is passed to an FB, but it won't work - you need to use the Index Register to access arrays passed as parameters to FBs.
Using the Index Register for this is described in one of the tips below.

```
FlagArray EQU F [10] ;array of Flags
...
CFB    0
       FlagArray     ;pass the array base to the FB
...
FB     0
Flag1  EQU =1        ;Wrong. Flag1 is =1, it is not FlagArray
Flag2  EQU Flag1+1   ;Wrong. Flag2 is =2, not FlagArray + 1
...
EFB
```

3) When [...] is used on an array symbol, it is assumed to be an array element reference, and not an array definition

It is not possible to create sub-arrays from an array, for example, this does not work:

```
BaseArray   EQU R 100 [10]
Array1      EQU BaseArray+0[5]  ;Array1 = R 105, it is NOT an array,
                                ;it's the same as BaseArray+0+5
Array2      EQU BaseArray+5[5]  ;Array2 = R 110, it is NOT an array,
                                ;it's the same as BaseArray+5+5
```

In fact, this will cause an "array bounds overflow" error for symbol Array2 because offset 10 is outside the array, offsets 0..9 are valid.

4) Do not use different symbol names for the same data
If using absolute addressing, make sure the address is assigned to (or derived from) one symbol in one place.
Using different symbol names for the same data will cause maintenance problems and bugs which are hard to find if you change one symbol but not the other.
Tip: Use the build option "Warn on symbols with same type and value", but note that this will also give warnings for array base address symbols with a zero offset.

5) Take care when using DEF, especially for FB parameters
Sometimes FB parameters are defined with DEF statements, because the same symbol can be re-defined again in the program without any errors.

```
FB     MyFB1
Run    DEF =1
Stop   DEF =2
Rewind DEF =3
...
EFB

FB     MyFB2
Stopp   DEF =1    ;unnoticed typing error
STH    Stop       ;error! this is the parameter for FB MyFB1
...
EFB
```

Now, in PG5 V2, you can use LEQU instead of DEF, or even the new $FBPARAM..$ENDFBPARAM directives.
Or use group names as described below.
This illustrates another common error:

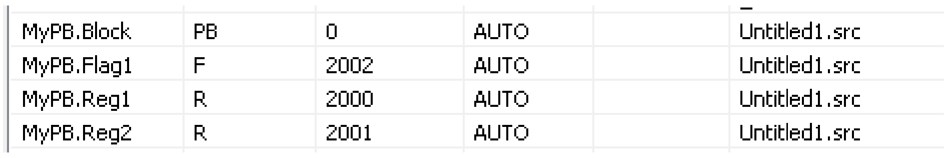
```
fbcall      EQU    FB
fbcallnew   EQU    FB
s1          EQU    F 1

            COB    0
                   0
            CFB    fbcall
                   s1
            ECOB

            FB     fbcall
int1        DEF    =1
int3        DEF    =1         ;different symbol for same parameter,
                              ;no warning!
            STH    int1
            OUT    int2       ;error, but no error message
            EFB

            FB     fbcallnew
int2        DEF    =1
            STH    int1       ;also an error, but no error message
            OUT    int2
            EFB
```

Note that symbols declared with DEF are not affected by $GROUP, but you can use a full symbol name, with a group part,
e.g. `MyBlock.Param1 DEF =1`

Use group names when defining associated symbols
Symbols which are from the same block can be given the same group name:

```
    $GROUP MyPB
    Block       EQU    PB
    Reg1        EQU    R
    Reg2        EQU    R
    Flag1       EQU    F

                PB     Block

                INC    Reg1
                INC    Reg2

                STH    Flag1

                EPB
    $ENDGROUP MyPB
```

This keeps all the symbols together in the Data List and Symbol Editor views:



MyPB.Block	PB	0	AUTO		Untitled1.src
MyPB.Flag1	F	2002	AUTO		Untitled1.src
MyPB.Reg1	R	2000	AUTO		Untitled1.src
MyPB.Reg2	R	2001	AUTO		Untitled1.src

Or you can use the group name as a kind of structure:

```
    $GROUP MyStruct
        Item1 EQU R
        Item1 EQU F
        Item3 EQU R
        Item4 EQU K 123
    $ENDGROUP MyStruct
```

To access a symbol outside the $GROUP, use '.' at the start
Sometimes the name of a symbol inside a group may be the same as a symbol outside. To be sure you are accessing the right symbol, precede it with a dot:

```
    Sym1    EQU R
    $GROUP Group0
    Sym1    EQU R      ;Group0.Sym1
    INC     Sym1       ;increments Group0.Sym1
    INC     .Sym1      ;increments Sym1
    $ENDGROUP
```

Use group names for SRXM/STXM data in the remote or slave PCD
Define symbols in other PCDs with a group name which is the name of the remote PCD.
In this way it will never be confused with local data:

```
    $GROUP Station100
    Symbol0 EQU R 100
    ...
    $ENDGROUP

    STXM    0
            1
            Symbol0               ;local data
            Station100.Symbol0    ;remote data
```

Initializing data with $INIT..$ENDINIT
Each block contains data which must be initialized before the block runs. This is often done in XOB 16. But it is not good practice to separate the initialization code from the rest of the block (bad "encapsulation"). To solve this problem, you can enclose the initialization code between $INIT..$ENDINIT, within the block itself. This code is placed at the start of XOB 16.
Note: This has the disadvantage that you cannot download in Run if the initialization code is changed because XOB 16 is not executed, see next tip.
See also Initializing Data.

```
    FB     MyFB
    Reg1   LEQU R
    Flag1  LEQU F
    $INIT
    ;Initialization code
    LD     Reg1
           123
    ACC    H     ;in case previous $INIT code reset it
    SET    Flag1
    $ENDINIT
    ;Rest of code
    ...
    EFB
```

To execute code the very first time a downloaded or restored-from-flash program is run
When a program is first downloaded or is restored from Flash, RAM Data Block values (DBs 4000 and above) are initialized with the values defined in the user program. This value, e.g. -1, can be checked by the user program, and then set to a different value. It will then only have the original value the very first time the program runs. For example:

```
;Original DB value after download is -1, after 1st run it is set to 0
FirstRun     EQU    DB 4000
DB FirstRun [1]     -1            ;DB has 1 element
TempReg      EQU    R             ;work register

             XOB    16
             ...
;Check for first time the program runs
             TFR    FirstRun      ;get DB element 0 into TempReg
                    K 0
                    TempReg
             INC    TempReg       ;increments -1 to 0 on first run,
                                  ;else increments 1 to 2
             ACC    Z             ;is result 0?
             JR     L NotFirstRun ;if no, then it is not the 1st run
             TFR    TempReg       ;set DB value to 0
                    FirstRun
                    K 0
```

```
;Code to be executed the first time the program runs goes here
            ...
NotFirstRun:
            ...
            EXOB
```

Initialize static data the first time a block is called

Instead of placing the initialization code in XOB 16 where it can be run only on start-up or restart cold, it can be coded inside the block itself and executed whenever a **volatile** Flag is found to be zero. You only need to reset this Flag to run the initialization code, and it is always run just once after power-up or a restart cold. If a first-time initialization value of 0 is used for the Flag, then it will be set to zero every time the block is downloaded.
This has the advantage that the initialization code can be changed and downloaded without needing a restart, and the init code and also be executed
See also Initializing Data.

```
    FB      MyBlock

   ;Initialize static data on first call
   ;volatile flag, always zero on start-up and on download
   ;with "first-time init data"
   MyBlock.InitFlag EQU F VOL := 0

    STH     MyBlock.InitFlag  ;has data been initialized yet?
    JR      H AlreadyInit     ;yes, skip initialization
    ACC     H                 ;no, set flag and do the initialization
    SET     MyBlockInitFlag
    ...                       ;do start-up initialization here

AlreadyInit:
    ...                       ;rest of program
    EFB
```

Segment directives

To insert code into a COB (task) or XOB (exception or interrupt handler), when the block is defined in another file, you can use the $COBSEG or $XOBSEG directives.
For example, this could be useful to define cyclic code to call a function and keep the call with the code that uses it:

```
    FB      CyclicFunc

    ;Call this function every 10 seconds fro COB 0
    $COBSEG 0
      STH   CyclicFuncCtr
      CFB   L CyclicFunc
    $ENDCOBSEG

    ;Reload the 10 second timer
    LD     CyclicFuncCtr
           T#10S

    ...

    EFB
```

Array size symbol

For every symbol which is defined as an array, another symbol is automatically generated which is assigned the size of array. The symbol name is prefixed with `_ArraySize_`.
For example:

```
    Symbol EQU R 10
```

This generates the internal symbol:

```
   _ArraySize_.Symbol EQU 10
```

This symbol can also be made Public:

```
   PUBL _ArraySize_.Symbol
```

Tip: You can see these in Project Manager's "Data List" view by checking the "Internal Symbols" option in the Data List Filter.

Texts/DBs 4000 and above are faster than Text/DBs 0..3999

Texts and DBs 4000 and above are accessed in a different way, and they are much faster.
The only problem is that they are in RAM, and could lose their values, whereas Texts/DBs 0..3999 can be in Flash or (E)EPROM.

Sharing Media between COBs and XOBs

FBs or PBs which contain "static" data (Registers, Flags etc whose values are retained between block calls) could run incorrectly if the block is called from more than one COB (task) or XOB (interrupt). It may be necessary to maintain different static data for each task or interrupt, otherwise it can be unexpectedly changed by a different task or by an interrupt.
Therefore blocks which are called from different COBs or XOBs should be carefully written so that static data is not shared, for example, use an FB parameter to pass the static data.
This can also cause serious problems when calling an FB or PB from a Fupla program with the Call FB or Call PB FBoxes..

Use the Index Register to access offsets from a base address (e.g. an array or I/O module)

Instead of passing many parameters to an FB call or macro, you can pass an array parameter, and use the Index Register to access the array.
Always save the original Index Register value, and restore it before returning from the FB.
For example:

```
    RegArray EQU R [10]
    ...
    CFB     IndexDemo
            RegArray    ;pass the base address
    ...

    FB      IndexDemo
    rSaveIndex TEQU R
    STI     rSaveIndex  ;save the index register
    SEI     K 0         ;start indexing from 0
    ...
    LDX     =1          ;use indexing instructions to access array
            0
    ...
    RSI     rSaveIndex  ;restore the index register
    EFB
```

Or you can use the parameter to load the Index Register, for example as the base address of an I/O

module.
LDL can be used to load a base address into a Register (for transfer to the Index Register), see "FB Parameters and LDL" below.

When to use Macros, FBs or PBs

There are three kinds of "blocks" you can use for creating functions or objects, each has different advantages and disadvantages.

Program Blocks (PBs)
PBs do not allow parameters, and always share the same data, no matter where they are called from. This makes them unsuitable for calling from more than one switchable COB task (or interrupt XOB), unless they have been specially programmed. The code of a PB exists only once. A PB is the same as an FB without parameters. (Actually, PBs are not very useful.)

Function Blocks (FBs)
FBs do allow parameters, but with certain restrictions (e.g. no 32-bit constants). Usually *all* the data used by the FB should be supplied as parameters, except temporary internal workspace data.
If static data is used then it has the same disadvantages as a PB - it may become unsuitable for calling from more than one switchable COB task (or interrupt XOB).
The code of an FB exists only once. An FB without parameters is the same as PB.

Macros
Macros can also have parameters, but these are not the same as FB parameters. A macro's parameter is just a string, and the string is used to replace the parameter reference in the code. Macro parameters do not have a type or a range, they can be anything at all, the only characters they cannot contain are comment characters '`;`', commas '`,`' or line feeds.

Macros are typically used to add the equivalent of new instructions, or to avoid repeating the same code many times but with slight differences.

If you ever find yourself copy/pasting the same code, then this could also be a good time to use a macro. Instead of repeating the same code many times, create a macro so that the code is defined just once, and call the macro with different parameters to generate the required code. Then if the code needs to be changed in the future, it can be done in just one place.

The other big advantage with macros is that you do not need to use the Index Register or Register Indirect instructions to access data from a base address. If the base address is passed as a macro parameter, you can access it directly. E.g.

```
    MyMacro MACRO ModuleBase
    STH    ModuleBase+0
    ANH    ModuleBase+1
    ANH    ModuleBase+2
    ANH    ModuleBase+3
    OUT    ModuleBase+4
    ENDM
```

If this was in an FB, you would need to use STHX etc.

There is no "call" instruction for a macro, so macros are much faster when used within loops.
Macros can call other macros too.The main disadvantage of a macro is that it generates more code.
Macros can significantly reduce code complexity and makes it easier to maintain.
See Macros.

Not enough Flags

Use the Bit Access Macros to access individual bits in Registers or Data Blocks.

Or use the Register Instructions (AND, OR etc) to test individual bits in Registers.

Not enough Registers

Use the DB Access Macros to access 32-bit values in Data Blocks, or use TFR, PUT, GET, COPY to transfer data between DBs and Registers.

FB parameters and LDL

If a parameter is used only in a LDL, the media type (mc) is removed from the call:

```
    CFB     0
            F 123    ;F is not needed, it is removed
    ....
    FB      0
    LDL     R 0      ;R 0 = 123
            =1
    ...
    EFB
```

But if it is accessed as a Flag, it will generate an error:

```
    FB      0
    LDL      R 0
            =1
    STH     =1       ;error!
    EFB
```

Use a Text to hold program information

At present, the PG5 does not store much information about the user program within the PCD itself.
The file name of the PCD file is stored in a DBX, but that's all.
However, you can easily create a Text which contains all the information you need, such as the revision number, release date, copyright notice etc.
This can be displayed using the Online Debugger (S-Bug) or easily displayed on a terminal or Web page.
Use a fixed text number, for example Text 0:

```
;Program Information, hard-wired in Text 0
TEXT 0  "Version: 123<CR>
        "Author: Me<CR><LF>
        "Copyright (C) Noware 2007<CR><LF>
        "Release date: 27th May 2007<CR><LF>
```

How to insert a macro parameter into a Text

Especially when writing FBoxes, it might be useful to fill a PCD text with, for example, the name of an FBox.
This can be achieved by using the ".s" postfix, which places the actual parameter string into the text, instead of its value.
For example:

```
    TextMacro MACRO param
      TEXT 100 "Hello ", param.s, "!"
    ENDM
    ...
    TextMacro(world)
```

The resulting text in the PCD will be:

```
    TEXT 100 "Hello world !"
```

<u>What is the difference between the media types "Constant" and "K Constant" ? (FAQ #100123)</u>

The 'K' is needed wherever an operand can be a data type (R T C etc) or a constant (an untyped number), so the interpreter knows what it is.
The main difference is the range of values for the two types. A K constant can be 0..16383, whereas 16-bit constants can be 0..65535 (unsigned) or -32768..+32767 (signed).

This is because of the structure of the 16-bit operand line. The data type (K) uses 2 bits, leaving 14 bits for the value.
This means that K constants can have a range 0..3FFF hex, which is 0..16383, and they are unsigned (can only be +ve).

```
    ADD   K 100   ;2 bits for type + 14 bits for value
          R 100   ;2 bits for type + 14 bits for address
          R 101   ;2 bits for type + 14 bits for address
```

Some instructions do not need a data type in the operand, and can be used with untyped constants, such as LD which allows a 32-bit untyped constant (signed or unsigned):

```
    LD    R 200
          21483647
```

LD actually uses two operand lines to hold hold the 32-bit value.
You can still use the 'K' type, but it is removed by S-Asm:

```
    LD    R 2000
          K 123     ;range limited to 0..16383
```

Untyped constants are normally used for loading Registers using LD.

K constants and 16-bit constants can be passed as FB parameters. If a constant without K is passed, S-Asm will add the K to the CFB call (adds the type bits 15 and 14), but only if the parameter is used only in instructions which need the 'K'. For example, this will not work:

```
    CFB   0
          123       ;not a K type constant
    ...
    FB    0
    ADD   R 0
          =1        ;this needs the K
          R 1
    LDL   R 2
          =1        ;this does not need the K
```

This cannot work because ADD needs to check bits 15 and 14 of the operand to get the type, but LDL would interpret bits 15 and 14 as part of the number.

<u>Why is the constant type in the assembled code different to my IL code ? (FAQ #100129)</u>

The assembler automatically adds (or removes) the K type from the generated code.
In PCDs instruction set there are two different types of constants: The normal "Constant" (16 or 32 bits, signed or unsigned) and the "K Constant" (14 bits, signed). See the previos tip for details.
Some instructions, like the load instructions LD, LDL and LDH, require a value without a type code.
Only a "Constant" can be used, the "K Constant" can't be used because the type bits (15 and 14) would be interpreted as part of the number.
Other instructions, like CMP and ADD, must have the type code because they can also access R and C types etc, and interpreter needs to know what it is.
Despite these rules, it is possible to use Constant or K types in your IL code as you want, because the assembler adds or removes the K depending on the instruction.

For example, the following IL code: After assembly it looks like this:

```
    CMP    R 0                         CMP    R 0
           0                                  K 0
    LD     R 4                         LD     R 4
           K 4                                4
```

This also works for CFB parameters. It is possible to pass a K constant as a parameter and use it with an instruction that needs an untyped Constant, or to pass a 16-bit untyped Constant and use it in an instruction which needs a K constant.

```
    CFB    0
           R 0
           1      ;no K, but param used in CMP - needs the K type
           K 2    ;has K, but param used in LDL - can't have the K type
```

The assembler replaces K 2 with 2 because the LDL instruction in the FB does not allow a K constant, and replaces 1 with K 1 because CMP needs the K:

```
    FB     0
    CMP    =1
           =2     ;K is required
    LDL    =1
           =3     ;K not allowed
```

This does not work if the parameter is used in both with-K and without K contexts. This will generate an error message "FB parameter has bad context", see preceding tip for an example.

Do not use slow instructions in XOB 0 (power down XOB) (FAQ #100066)

XOB 0 has max. 10 milliseconds to execute before power fails. Some instructions can take longer than this, so they cannot be used in XOB 0. For example,
SYSWR 2000..2049 : Write nonvolatile register (user EEPROM)
SYSWR 3000..3001 : Flash copy/erase
SYSWR 3100..3102 : Flash copy/erase

How can I read the PCD's serial number from the user program? (FAQ #100834)

There is a System Function call to do this, which returns the 32-bit serial number in a Register.
In S-Edit, open the Function Selector window and open the "SF System Library", select the "ReadSerialNum" function.
If you press F1 it shows the help for this function. If you double-click on it, the call is inserted into the IL code:

```
    CSF    S.SF.SYS.Library         ;Library number
           S.SF.SYS.ReadSerialNum   ;Read PCD serial number into Register
                                    ;1 R OUT, R to receive PCD's serial number
```

Fill in the register number, and check the Error flag incase the System Function is not supported:

```
    CSF    S.SF.SYS.Library         ;Library number
           S.SF.SYS.ReadSerialNum   ;Read PCD serial number into Register
           R 100                    ;1 R OUT, R to receive PCD's serial number
    JR     E NotSupported
```

(Note: This needs the library's include file to be include in the source file, so the symbols are defined:

```
    $include "<$LibsDir>\SF\SFSysLib_en.lib"
```

S-Edit now does this automatically, so you do not need to add this yourself if you use the Function Selector.)

How to copy Text into another Text (FAQ #100886)
There are now a System Function calls for copying Texts and Data Block data.
In S-Edit, open the Function Selector window and look at the "SF DB Access Library". This also contains functions for copying Texts, which also supports the @ and $ formats.
Select a function and press DF1 to get help on the function.
Note: These functions are only for NT systems or PCD1/PCD models with the latest firmware.
If your firmware version does not support the System Function call, the Error flag will be set.
Minimum FW versions:
D1.M1x5 $A5
D2.M150 $D1
Dx.M170 $21
S1.C6/C8 $B2
D2.M480 $29
D3.Mxxxx $25

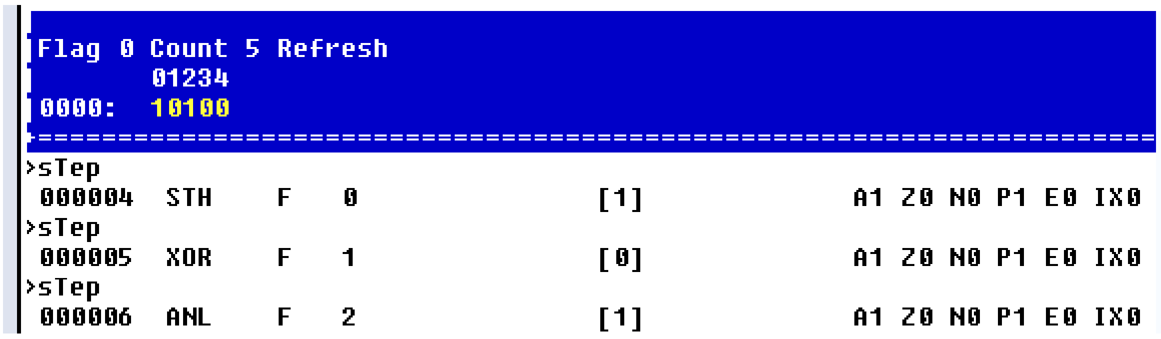
Example:
Source Text:
```
TEXT 100 "Alarm on station 10, @L0100. Motor over temperature: $R0110°C.<CR><LF>"
"Please call $L0020"
```

Register 100 has the value 2
Register 0110 has the value 220
```
Text 2 "Section B"
Text 20 "John on 044 345 32 32"
```

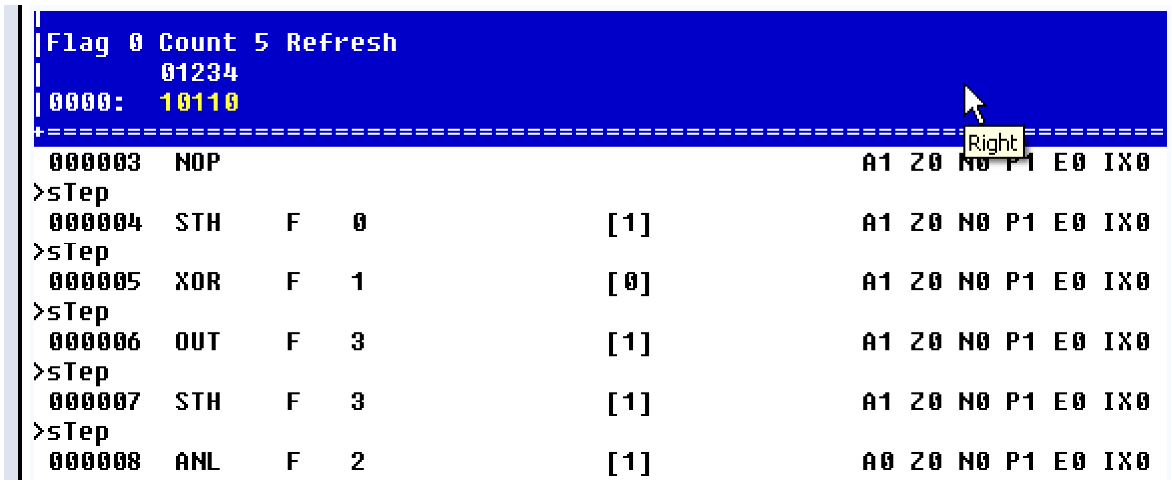
The Register 100 contains the pointer to the sub-text.
If the Register 100 has the value 2 then the text 2 is inserted on the position @L0100.
After copying the source text to the destination text, the destination text will be the following:
```
"Alarm on station 10, Section B. Motor over temperature: 264°C. Please call John on
```

Is it possible to search an expression within a PCD text? (FAQ #101186)
Can I read a value from a PCD text and copy it into a register? (FAQ #101187)
Yes, see the "SF DB Access Library", described in the preceding tip.

XOR and OR calculations in IL - surprising results (FAQ #100720)
You may wonder about surprising results when programming in IL with XOR and OR operations.
It's good to know the philosophy behind the behaviour of the ACCU.
Experienced programmers may wonder why the `XOR F 1` instruction in the example below sets the ACCU to 0 (F 0=1 and F 1=0, so 1 XOR 0 should be 1).
And why the `ANL F 2` instruction sets the ACCU to 1 (Accu=0 and F 2=1, 0 AND NOT 1 should be 0).


```
Flag 0 Count 5 Refresh
        01234
0000:   10100
==========================================================================
>sTep
 000004   STH     F    0               [1]            A1 Z0 N0 P1 E0 IX0
>sTep
 000005   XOR     F    1               [0]            A1 Z0 N0 P1 E0 IX0
>sTep
 000006   ANL     F    2               [1]            A1 Z0 N0 P1 E0 IX0
```

The reason is that XOR and OR operations work like "open parentheses" for logic calculations. If you want to apply the result of the XOR and OR operation, you have to "close parentheses" by programming an OUT instructions on a dummy Flag and testing the state of this Flag using STH, as shown in the example below.



```
Flag 0 Count 5 Refresh
        01234
0000:   10110
+=====================================================================
 000003  NOP                                           A1 Z0 N0 P1 E0 IX0
>sTep
 000004  STH    F   0          [1]                     A1 Z0 N0 P1 E0 IX0
>sTep
 000005  XOR    F   1          [0]                     A1 Z0 N0 P1 E0 IX0
>sTep
 000006  OUT    F   3          [1]                     A1 Z0 N0 P1 E0 IX0
>sTep
 000007  STH    F   3          [1]                     A1 Z0 N0 P1 E0 IX0
>sTep
 000008  ANL    F   2          [1]                     A0 Z0 N0 P1 E0 IX0
```

`OUT F 3` gives the correct result for 1 XOR 0. Another OUT instruction after `ANL F 2` would produce the correct result 0 (0 AND NOT 1 = 0).

Handling arrays as FB parameters (FAQ #100724)

When an array is passed to an FB, the array cannot be accessed with code like this:

```
FlagArray  EQU F [10] ;array of Flags
...
CFB     0
        FlagArray     ;pass the array base to the FB
...
FB      0
Flag1   EQU =1        ;Wrong. Flag1 is =1, it is not FlagArray
Flag2   EQU Flag1+1   ;Wrong. Flag2 is =2, not FlagArray + 1
...
EFB
```

To access an array from inside the FB, you should use the Index Register as in this example:

```
FlagArray EQU F [3]
...
CFB     1
        FlagArray
...
FB      1

rSaveIndex LEQU R
STI     rSaveIndex  ;save the Index Register

Flag1   EQU =1
STH     Flag1       ;same STH FlagArray

SEI     K 1         ;Index register is 1
STHX    Flag1       ;same as STH FlagArray+1
SEI     K 2         ;Index Register is 2
```

```
STHX    Flag1       ;same as STH FlagArray+2
...
SEI     rSaveIndex  ;restore the Index Register before returning
EFB
```

How to change the base of a logarithm (FAQ #101238)

The Saia PG5 instruction set does support the natural logarithm Ln (hyperbolic, base e, floating point instruction FLN).
Logarithms to any other base like 10 (Log10) can be calculated with the method described below.
The Ln of x can be divided by the Ln of the desired base: Log n (x) = Ln(x) / Ln(n)
Example in case the logarithm with the base 10 is needed:
Log10(x) = Ln(x) / Ln(10)
The value of Ln(10) (=2.302585093) can be stored in a constant or variable so it does not need to be recalculated each time a Log10 is required.

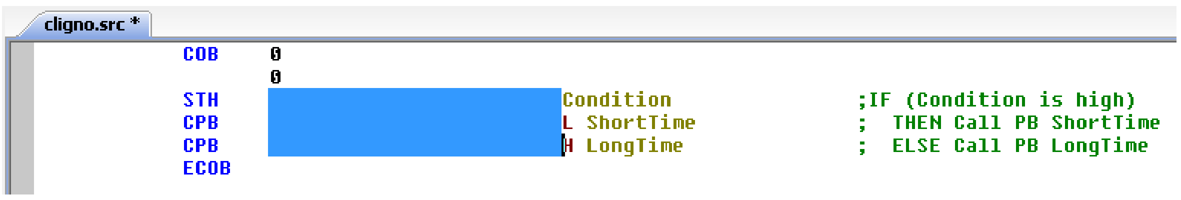
Block select mode in S-Edit (marking columns of text)

The IL Editor S-Edit has a "block select mode" which allows selecting columns instead of lines of code.

1. Hold down the Alt key.
2. Keeping the Alt key down, click the left button at the start of the text.
3. Drag the mouse cursor to the end and release the left button.

or

1. Click on the left-hand mouse top left-hand corner of the column you want to mark, or move the caret there with the cursor keys.
2. Hold down the Shift and Alt keys.
3. Click the left-hand mouse button on the lower right-hand corner of the area to be marked - the first click marks the lines.
4. Click a second time - the second click marks the column.



```
COB     0
        0
STH                 Condition          ;IF (Condition is high)
CPB                 L ShortTime        ;  THEN Call PB ShortTime
CPB                 H LongTime         ;  ELSE Call PB LongTime
ECOB
```

Block select mode in S-Edit

How to get a data address at run time

If you need to load the address of a item into a Register, such as a Flag address, another Register's address, or even a block number, you can simply load the symbol's value into a Register using the LD instruction. The symbol's type is ignored. This also works for dynamic addresses and Externals. For example:

```
Symbol  EQU R 123
Block   EQU FB 10
EXTN ExternalSym
AddsReg EQU R

LD      AddsReg
        Symbol       ;AddsReg = 123
LD      AddsReg
        Block        ;AddsReg = 10
```

```
LD      AddsReg
        ExternalSym  ;AddsReg = ExternalSym
```

How to use $USE / $IFUSED

These directives can be used to solve the problem that $IF conditional assembly directives cannot reference External symbols.
$IF statements cannot reference Externals because the value must be known at assembly time, and Externals are only resolved at link time.
See the description in this help file: $USE, $IFUSED, $INUSED.
If you are still confused, see the next tipple.

2 Bit Instructions

Bit instructions work with the Accumulator, Inputs, Outputs, Flags and the state of Timers or Counters.

STH	Start High
STL	Start Low
ANH	And High
ANL	And Low
ORH	Or High
ORL	Or Low
XOR	Exclusive Or
ACC	Accumulator Operations
DYN	Dynamic (edge detection)
OUT	Set Element from ACCU
SET	Set Element
RES	Reset Element
COM	Complement Element
SETD	Set Element Delayed
RESD	Reset Element Delayed

2.1 STH - Start High

Description
The ACCU is set to the logical state of the addressed element. This is the start of a new linkage line. The previous linkage results are cleared with the start instruction; simultaneously the signal state "H" of the addressed element I, O, F, T, C will be read and the result stored in the ACCU.

Format
```
STH[X]  [=] element (i)   ;I O F T C
```

Example
```
STH     I 7  ;ACCU := state of Input 7
```

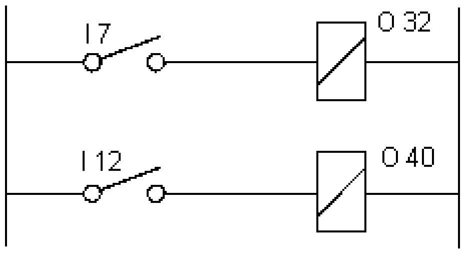
Flags

ACCU	Set to the state of the addressed I O F T or C
Status Flags	Unchanged

See also
STHS, STL

Note
If a Timer or Counter contains 0 its state is Low, otherwise its state is High.

Practical example



```
    ;A minimum program in the PCD must consist of one COB
    COB     0       ;start of COB
            0
    STH     I 7     ;if Input 7 is High
    OUT     O 32    ;then set Output 32
                    ;else reset Output 32
    STH     I 12    ;if Input 12 is Hight
    OUT     O 40    ;then set Output 40
                    ;else reset Output 40
    ECOB            ;end of COB
```

2.2 STL - Start Low

Description
The ACCU is set to the inverted logical state of the addressed element. This is the start of a new linkage line.
The previous linkage results are cleared with the start instruction; simultaneously the signal state "L" of the addressed element I, O, F, T, C will be read, inverted and the result stored in the ACCU.

Format
```
STL[X]  [=] element (i)   ;I O F T C
```

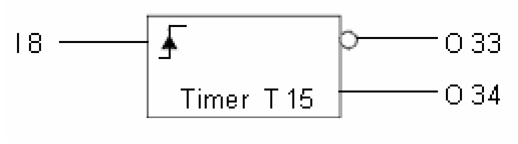
Example
```
STL    I  9  ;ACCU = inverted state of Input 9
```

Flags

ACCU	Set to the inverted state of the addressed I O F T or C
Status Flags	Unchanged

See also
STH

Practical example



```
    COB     0       ;start of COB
            0
    STH     I 8     ;if Input 8 goes High
    DYN     F 10    ;(DYN detects rising edge)
    LD      T 15    ;then load Timer with 2 sec
            20      ;(20 x 100ms)
    STL     T 15    ;if the time has elapsed
    OUT     O 33    ;then set Output 33
```

```
                        ;else reset Output 33
    STH     T 15        ;if the time has not elapsed
    OUT     O 34        ;then set Output 34
                        ;else reset Output 34
    ECOB                ;end of COB
```

2.3 ANH - And High

Description
The ACCU is AND linked with the logical state of the addressed element and the ACCU is set to the result.

ACCU	State	Result
L	L	L
L	H	L
H	L	L
H	H	H

Format
```
ANH[X]    [=] element (i)   ;I O F T C
```

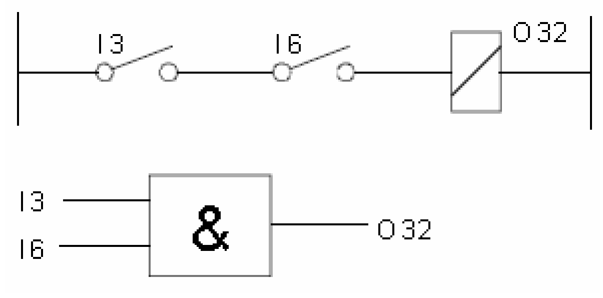
Example
```
ANH     I 3    ;ANDs the ACCU with the state of Input 3
ANHX    I 128  ;ANDs the ACCU with Input (128 + Index)
```

Flags

ACCU	Set according to the result
Status Flags	Unchanged

See also
ANL

Practical example



```
    COB     0           ;start of COB
            0
    STH     I 3         ;if Input 3 is High
    ANH     I 6         ;and Input 6 is High
    OUT     O 32        ;then set Output 32
                        ;else reset Output 32
    ECOB                ;end of COB
```

2.4 ANL - And Low

Description
The ACCU is AND linked with the inverted logical state of the addressed element, the ACCU is set to the result.

ACCU	State	Result
L	L	L
L	H	L
H	L	H
H	H	L

Format
```
ANL[X]     [=] element (i)    ;I O F T C
```

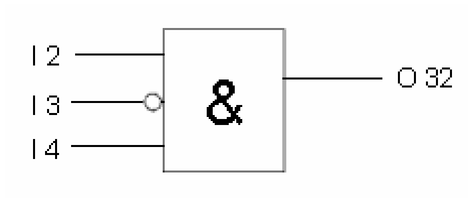
Example
```
ANL    I 4     ;ANDs the ACCU with inverted state of Input 4
ANHX   I 128   ;ANDs the ACCU with inverted Input (128+Index)
```

Flags

ACCU	Set according to the result
Status Flags	Unchanged

See also
ANH

Practical example



```
    COB     0       ;start of COB
            0
    STH     I 2     ;if Input 2 is High
    ANL     I 3     ;AND Input 3 is Low
    ANH     I 4     ;AND Input 4 is High
    OUT     O 32    ;then set Output 32
                    ;else reset Output 32
    ECOB            ;end of COB
```

2.5 ORH - Or High

Description
The ACCU is OR linked with the logical state of the addressed element, and the ACCU is set to the result.
OR instructions are used for parallel linkages of elements.

The main linkage begins with a start instruction (STH or STL). Each additional parallel partial linkage begins with an ORH.
If a parallel linkage is successful (ACCU=High), then the logical states of all the following partial linkages no longer exercise any influence on the result of the total linkage.

ACCU	**State**	**Result**
L	L	L
L	H	H
H	L	H
H	H	H

Format

```
ORH[X]    [=] element (i)    ;I O F T C
```

Example

```
STH     I 5     ;if Input 5 is High
ORH     I 13    ;or Input 13 is High
                ;then ACCU = 1, else ACCU = 0
```

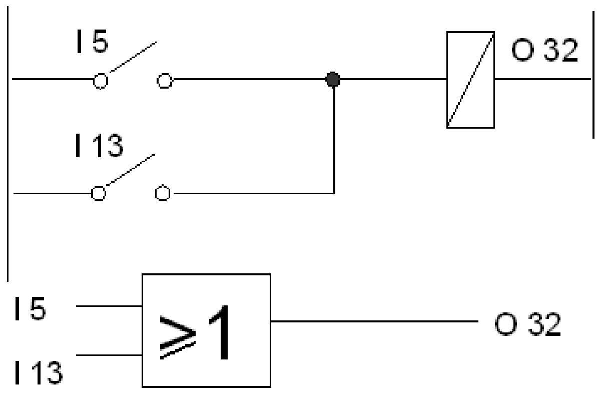
Flags

ACCU	Set according to the result
Status Flags	Unchanged

See also

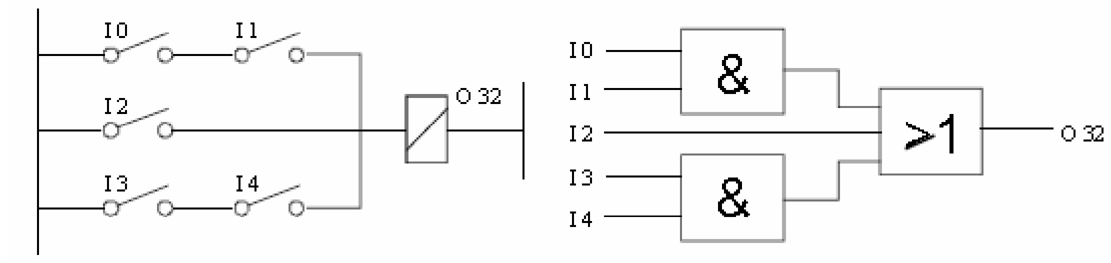
ORL

Practical example 1



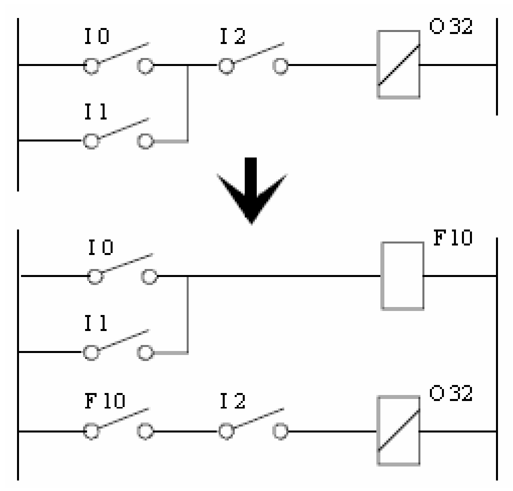
```
    COB       0       ;start of COB
              0
    STH       I 5     ;if Input 5 is High
    ORH       I 13    ;or Input 13 is High
    OUT       O 32    ;then set Output 32
                      ;else reset Output 32
    ECOB              ;end of COB
```

Practical example 2



```
COB     0       ;start of COB
        0
STH     I 0     ;if Input 0 is High
ANH     I 1     ;and Input 1 is High
ORH     I 2     ;or Input 2 is High
ORH     I 3     ;or Input 3 is High
ANH     I 4     ;and Input 4 is High
OUT     O 32    ;then set Output 32
                ;else reset Output 32
ECOB            ;end of COB
```

It can be seen from the above example that the OR instruction has priority over AND.



```
COB     0       ;start of COB
        0
STH     I 0     ;if Input 0 is High
ORH     I 1     ;or Input 1 is High
OUT     F 10    ;then set Flag 10
                ;else reset flag 10
STH     F 10    ;if Flag 10 is High
ANH     I 2     ;and Input 2 is High
OUT     O 32    ;then set Output 32
                ;else reset Output 32
ECOB            ;end of COB
```

2.6 ORL - Or Low

Description

The ACCU is OR linked with the inverted logical state of the addressed element, and the ACCU is set to the result.
OR instructions are used for parallel linkages of elements.See ORH for details.

ACCU	State	Result
L	L	H
L	H	L
H	L	H
H	H	H

Format

```
ORL[X]     [=] element (i)    ;I O F T C
```

Example

```
STH    I 3    ;if Input 3 is High
ORL    I 7    ;or Input 7 is Low
              ;then ACCU = 1, else ACCU = 0
```

Flags

ACCU	Set according to the result
Status Flags	Unchanged

See also

ORH

2.7 XOR - Exclusive OR

Description

The ACCU is XOR linked with the logical state of the addressed element and the ACCU is set to the result.
XOR can be used to compare the states of two elements.
If they are the same the ACCU is set Low; if they are different it is set High.

Note: To follow XOR with an ANH/ANL instruction, first end the sequence with an OUT to store the XOR result, then start a new linkage with STH/STL. An AND linkage after XOR will cause "Warning 22: Ignoring AND after XOR instruction".

ACCU	State	Result
L	L	L
L	H	H
H	L	H
H	H	L

Format

```
XOR[X]    [=] element (i)    ;I O F T C
```

Example

```
XOR   I 5    ;ACCU = ACCU XOR Input 5
```

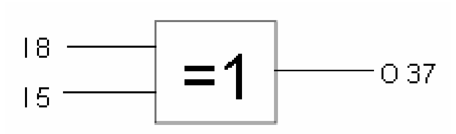
Flags

ACCU	Set according to the result
Status Flags	Unchanged

See also

OR
AND

Practical example



```
    COB     0       ;start of COB
            0
    ;if Input 8 is High and Input 5 is Low
    ;or Input 8 is Low and Input 5 is High
    STH    I 8    ;O 37 = I 8 XOR O 37
    XOR    I 5
    OUT    O 37   ;then set Output 37
                  ;else reset Output 37

    ECOB          ;end of COB
```

2.8 ACC - Accumulator Operations

Description
Modifies the state of the Accumulator according to the code:
The operands cannot be supplied as a Function Block parameters.

C	Complement	ACCU is complemented (inverted)
H	High	ACCU is set High (1)
L	Low	ACCU is set Low (0)
P	Positive	ACCU is set to Positive (P) flag state
N	Negative	ACCU is set to Negative (N) flag state
Z	Zero	ACCU is set to Zero (Z) flag state
E	Error	ACCU is set to Error (E) flag state

Format
```
ACC    code     ;code = C | H | L | P | N | Z | E
```

Example
```
ACC    H        ;sets ACCU to 1
ACC    E        ;sets ACCU to state of E status flag
```

Flags

ACCU	Set according to the result
Status Flags	Unchanged

See also
OUT,
Condition Codes

Practical example

```
    CMP    R 99    ;compare R 99
           'x'     ;with character 'x'
    ACC    Z       ;if equal then ACCU is set High
                   ;(copy Z status flag to the ACCU)
    ...
```

2.9 DYN - Dynamic Edge Detection

Description
For rising or falling edge detection.
The result in the ACCU is High only when the ACCU goes from Low to High on consecutive executions of DYN (rising edge).
The Flag given in the operand stores the previous state of the ACCU.
If the ACCU is Low, it remains Low, and the Flag is also set Low. The Flag need not be Low the first

time DYN is executed.
For rising edge detection, use STH to interrogate the element; for falling edge detection, use STL.

Format

```
DYN[X]   [=] flag (i)   ;F
```

Example

```
DYN      F 100      ;Flag 100 stores dynamic ACCU state
```

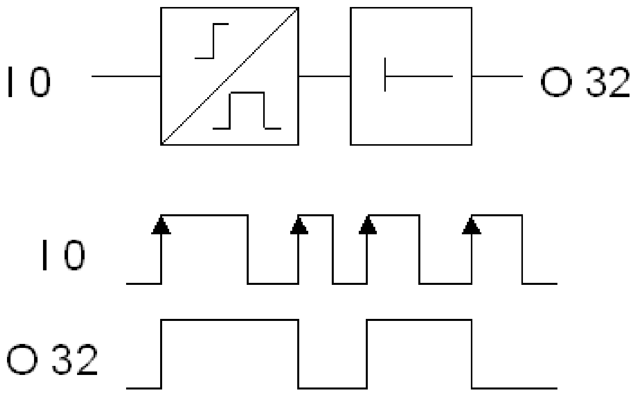
Flags

ACCU	Set High on rising edge
Status Flags	Unchanged

See also

STH
STL
ANH
ANL
ORH
ORL

Practical example



```
;Solution with DYN instruction
COB    0      ;start of COB
       0
STH    I 0    ;if Input 0 goes High
DYN    F 500  ;(edge detection)
COM    O 32   ;then toggle Output 32
              ;else do nothing
ECOB          ;end of COB

;Solution without DYN instruction
COB    0      ;start of COB
       0
STH    I 0    ;if Input 0 is High
ANL    F 500  ;and Flag 500 is Low
SET    F 500  ;then set Flag 500 to High
COM    O 32   ;invert Output 32
              ;else do nothing
STL    I 0    ;if Input 0 is Low
RES    F 500  ;then reset Flag 500 (state = L)
              ;else do nothing
```

```
    ECOB            ;end of COB
```

2.10 OUT - Set Element From Accumulator

Description
Sets an Output or Flag to the state of the ACCU.
If the ACCU is High then the Output or Flag is set High.
If the ACCU is Low, then the Output or Flag is set Low.

Format
```
OUT[X]    [=] element (i)   ;O F
```

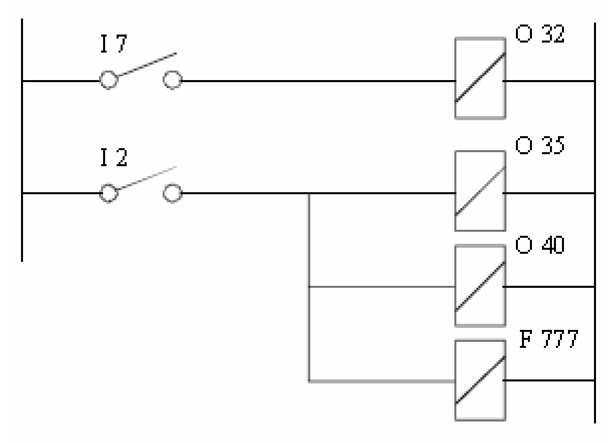
Example
```
OUT      O 32   ;sets output 32 to the state of the ACCU
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
OUTL
OUTS

Practical example



```
    COB     0       ;start of COB
            0
    STH     I 7     ;if Input 7 is High
    OUT     O 32    ;then set Output 32
                    ;else reset Output 32
    STH     I 2     ;if Input 2 is High
    OUT     O 35    ;then set Output 35, else reset Output 35
    OUT     O 40    ;and set Output 40, else reset Output 40
    OUT     F 777   ;and set Flag 777, else reset Flag 777
    ECOB            ;end of COB
```

2.11 OUTL - Set Element From Inverted Accumulator

Description
Sets an Output or Flag to the inverted state of the ACCU.
If the ACCU is High then the Output or Flag is set Low.
If the ACCU is Low, then the Output or Flag is set High.

Format
```
OUTL[X]   [=] element (i)    ;O F
```

Example
```
OUTL    O 32    ;Output 32 = inverted state of the ACCU
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
OUT
OUTS

2.12 SET - Set Element

Description
The Output or Flag is set High only if the ACCU is High.
If the ACCU is Low, nothing is done.
An Output or Flag set with a SET-instruction remains set (High) until it is reset again by a RES instruction.
This instruction is only executed if the ACCU is High.

Format
```
SET[X]     [=] element(i)    ;O F
```

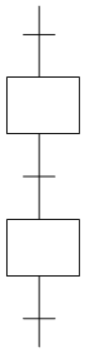
Example
```
SET    O 32    ;if ACCU is H then set Output 32
```

Flags

ACCU	Unchanged This instruction is executed only if the ACCU is High.
Status Flags	Unchanged

See also
RES
SETD
RESD

Practical example
Graftec program. Outputs 36 and 37 must blink after Input 7 has been switched on.



```
STH   I 7     ; is input = 1 ?

SET   O 36    ; set output 36
SET   O 37    ; set output 37
LD    T 1     ; load timer 1
      5       ; with 0.5s

STL   T 1     ; timer elapsed ?

RES   O 36    ; reset output 36
RES   O 37    ; reset output 37
LD    T 1     ; load timer no. 1
      10      ; with 1 second

STL   T 10    ; has timer elapsed?
```

2.13 RES - Reset Element

Description
The Output or Flag is set Low only if the ACCU is High. If the ACCU is Low, nothing is done.

Format
```
RES[X]   [=] element (i)  ;O F
```

Example
```
RES    O 13  ;if ACCU is High then reset Output 13
```

Flags

ACCU	Unchanged This instruction is executed only if the ACCU is High.
Status Flags	Unchanged

See also
SET
SETD
RESD

Practical example
See SET

2.14 COM - Complement Element

Description
The state of the Output or Flag is complemented (inverted) only if the ACCU is High. If the ACCU is Low, nothing is done.

Tips: To be sure this instruction is executed, precede it with ACC H.
This instruction can be used to trigger the Watchdog (Output 255) from a cyclic program, e.g.
```
    ACC   H
    COM   O 255
```

Format

```
COM[X]    [=] element (i)    ;O F
```

Example

```
COM    O 32   ;invert the state of Output 32 if ACCU is High
```

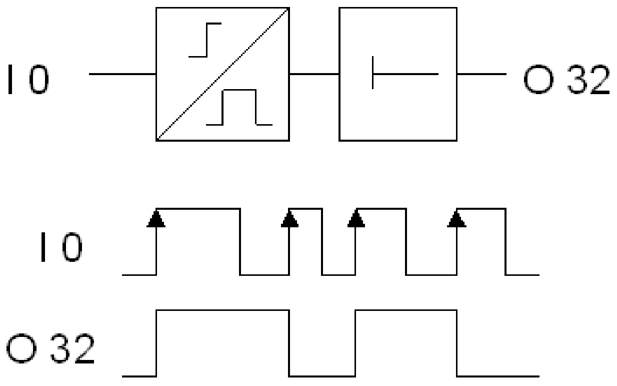
Flags

ACCU	Unchanged This instruction is executed only if the ACCU is High.
Status Flags	Unchanged

See also

OUT
SET
RES
DYN

Practical example



```
    COB      0        ;start of COB
             0
    STH      I 0      ;if Input 0 goes High
    DYN      F 500    ;(edge detection)
    COM      O 32     ;then complement Output 32
                      ;else do nothing
    ECOB              ;end of COB
```

2.15 SETD - Set Element Delayed

Description

The Output or Flag is set High after the delay given in the 2nd operand only
if the ACCU is High. The delay is in timebase units, as set by the DEFTB instruction.

A maximum of 16 delayed instructions can be in progress at the same time.
The operands cannot be supplied as Function Block parameters.

Format

```
SETD[X]   element (i)   ;O F
          delay         ;delay in timebase units
```

Example

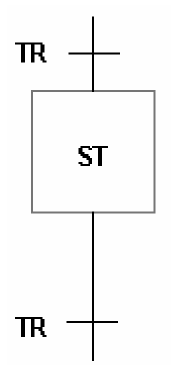
```
SETD    O 32    ;if ACCU is High then set Output 32
        100     ;after 100 x 100ms = 10 seconds
```

Flags

ACCU	Unchanged
Status Flags E	Set if more than 16 delayed actions are attempted

See also
RESD
DEFTB

Practical example
Graftec program.



```
STH     ...    ;condition to continue

SET     O 35   ;Output 35 is set immediately
SETD    O 40   ;Output 40 is set after 120 seconds
        1200   ; independently of the running
               ; Graftec program


STH     ...   ';condition to continue
```

2.16 RESD - Reset Element Delayed

Description
The Output or Flag is set Low after the delay given in the 2nd operand only if the ACCU is High.
The delay is in timebase units, as set by the DEFTB instruction.

A maximum of 16 delayed instructions can be in progress at the same time.
The operands cannot be supplied as Function Block parameters.

Format
```
RESD[X]    element (i)   ;O F
           delay         ;delay in timebase units
```

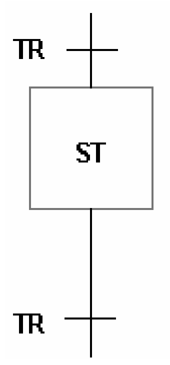
Example
```
RESD     O 32     ;if ACCU is Hight then reset Output 32
         100      ;after 100 x 100ms = 10 seconds
```

Flags

ACCU	Unchanged
Status Flags E	Set if more than 16 delayed actions are attempted

See also
SETD
DEFTB

Practical example
Graftec program.



```
STH     ...    ;condition to continue

SET     O 35   ;Output 35 is set immediately
SETD    O 35   ;Output 35 is reset after 5 seconds
        1200   ; independently of the running
               ; Graftec program


STH     ...    ;condition to continue
```

3 Register Instructions

These instructions transfer data to or from Registers.
Registers can contain binary, decimal, BCD or floating point values.

Loading Data

LD	Load 32-bit value
LDL	Load low word, lower 16 bits
LDH	Load high word, upper 16 bits

Primary arithmetic

INC	Increment Register
DEC	Decrement Register

Note: For arithmetic with floating point values, the floating point instructions must be used.

Moving Data

MOV	Move data	
COPY	Copy data	} Specially useful
GET	Get data	} for indexed
PUT	Put data	} addressing
TFR	Transfer data	
TFRI	Transfer data indirect	

Binary Input/Output

BITI	Bit in
BITIR	Bit in reversed
BITO	Bit out
BITOR	Bit out reversed

BCD Digit Input /Output

DIGI	Digit in
DIGIR	Digit in reversed
DIGO	Digit out
DIGOR	Digit out reversed

Logical

AND	AND Registers
OR	OR Registers
EXOR	Exclusive-OR Registers
NOT	Complement Register

Rotates and Shifts

SHIU	Shift Registers up
SHID	Shift Registers down
ROTU	Rotate Registers up
ROTD	Rotate Registers down
SHIL	Shift Register left
SHIR	Shift Register right
ROTL	Rotate Register left
ROTR	Rotate Register right

3.1 AND - And Registers

Description

The contents of the 1st Register is logically ANDed with the contents of the second Register, and the result is placed in the 3rd Register.

Format

```
AND[X]  [=] value1 (i)  ;R
        [=] value2      ;R
        [=] result (i)  ;R
```

Example

```
AND     R 11            ;AND Register 11 with
        R 12            ;with Register 12
        R 13            ;and put the result in Register 13
```

R 13 contains a 1 bit for every bit which is a 1 in both R 11 AND R 12.

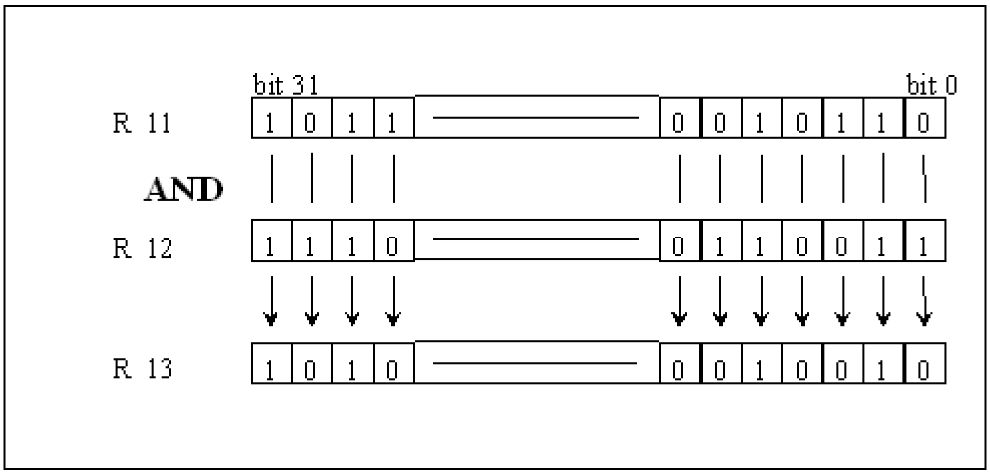
Flags

ACCU		Unchanged
Status Flags	E	Always set Low
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

OR
NOT
EXOR

Practical example



3.2 BITI - Bit In

Description

Moves a number of binary bits from Inputs, Outputs, Flags, a Timer or a Counter into a Register.
    The 1st operand is the number of bits to be moved (1..32).
    The 2nd operand is the source (I, O, F, T or C).
    The 3rd operand is the destination Register.
If the source is Inputs, Outputs or Flags, the source address given is the lowest address of the range.
The LOWEST address becomes the LEAST SIGNIFICANT bit in the destination Register.

Format
```
BITI[X] [=] bits       ;number of bits to read 1..32
        [=] source     ;source I O F T C
        [=] dest {i}   ;destination R
```

Example
```
BITI    16             ;read 16 bits
        I 32           ;from Inputs 32..47
        R 10           ;and store in Register 10 bits 0..15
```

Flags

ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the value read
	Z	Set according to the value read
	N	Set according to the value read

See also
BITIR
DIGI
DIGIR

Practical example
When input 8 goes High, an 8-bit binary value is read from inputs 0 to 7 and stored in Register 500.

```
    COB     0        ;start of COB
            0
    ...
    STH     I 8      ;if Input 8 goes High
    DYN     F 100    ;(uses F 100 to detect the change)
    JR      L Next   ;jump if not gone Low to High
    BITI    8        ;read 8 bits
            I 0      ;from Inputs 0..7
            R 500    ;and store in R 500
Next:
    ...
    ECOB
```

3.3 BITIR - Bit In reversed

Description
Moves a number of binary bits from Inputs, Outputs, Flags, a Timer or a Counter into a Register.
The 1st operand is the number of bits to be moved (1..32).
The 2nd operand is the source (I, O, F, T or C).
The 3rd operand is the destination Register.
If the source are Inputs, Outputs or Flags, the source address is the lowest address of the range.
The LOWEST address becomes the MOST SIGNIFICANT bit in the destination Register.

Format
```
BITIR[X]  [=] bits       ;number of bits to read 1..32
          [=] source     ;source I O F T C
          [=] dest (i)   ;destination R
```

Example
```
BITIR     16             ;read 16 bits
```

```
          I 32          ;from Inputs 32..47
          r 10          ;and store them in Register 10 bits 15..0
```

Flags

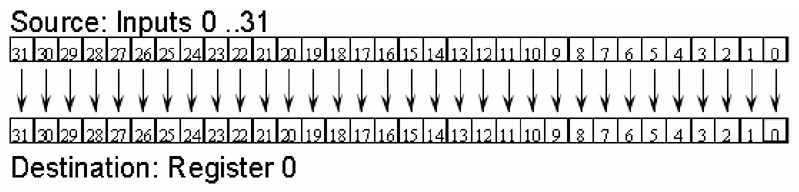
ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the value read
	Z	Set according to the value read
	N	Set according to the value read

See also

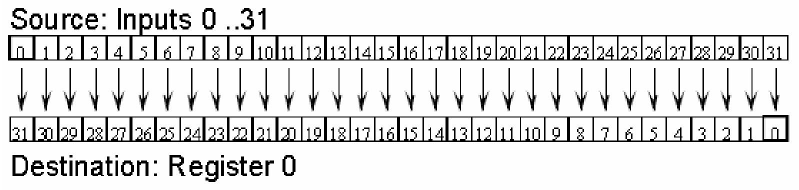
BITI
DIG
DIGIR

Practical example

```
BITI    32
        I 0
        R 0
```



```
BITIR   32
        I 0
        R 0
```



3.4 BITO - Bit Out

Description

Moves a number of binary bits from a Register to Outputs or Flags, or to bits in a Timer or Counter.
The 1st operand is the number of bits to transfer (1..32).
The 2nd operand is the source Register number.
The 3rd operand is the destination Outputs, Flags, Timer or Counter.
If the destination is Outputs or Flags, the destination address is that of the first address of the range.
The LEAST SIGNIFICANT bit of the Register is moved to the LOWEST address.

Format

```
BITO[X]   [=] bits        ;number of bits to move 1..32
          [=] source (i)  ;source R
          [=] dest        ;destination O F T C
```

Example

```
BITO      8               ;move 8 bits
          R 10            ;from Register 10 bits 0..7
          O 48            ;to Outputs 48..55
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
BITOR
DIGO
DIGOR

Practical example
Copy the states of Inputs 0..15 to Outputs 32..47.

```
    COB     0        ;start COB
            0
    BITI    16       ;read 16 bits
            I 0      ;from Inputs 0..15
            R 0      ;to Register 0 bits 0..15
    BITO    16       ;write 16 bits
            R 0      ;from Register 0
            O 32     ;to Outputs 32..47
    ECOB
```

3.5 BITOR - Bit Out Reversed

Description
Moves a number of binary bits from a Register to Outputs, Flags or bits in a Timer or Counter.
The 1st operand is the number of bits to transfer (1..32).
The 2nd operand is the source Register number.
The 3rd operand is the destination Outputs, Flags, Timer or Counter.
If the destination is Outputs or Flags, the destination address is that of the lowest element in the range.
The LEAST SIGNIFICANT bit of the Register is moved to the HIGHEST address.

Format
```
BITOR[X]   [=] bits        ;number of bits to move 1..32
           [=] source (i)  ;source Register R
           [=] dest        ;destination O F T C
```

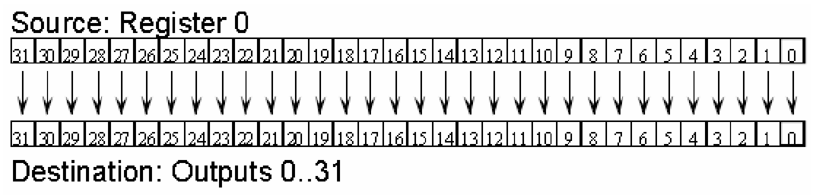
Example
```
BITOR     8       ;move 8 bits
          R 10    ;from Register 10 bits 0..7
          O 48    ;to Outputs 55..48
```
Flags

ACCU	Unchanged
Status Flags	Unchanged

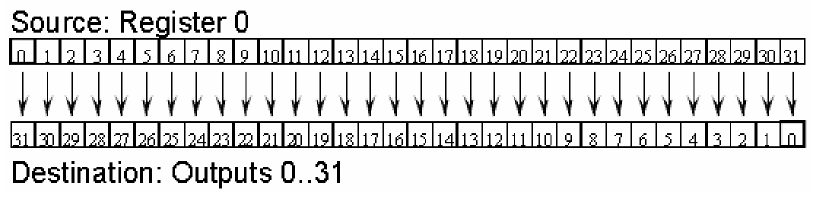
See also
BITO
DIGO
DIGOR

Practical example

```
BITO    32
        R 10
        O 32
```



```
BITO    32
        R 10
        O 32
```



3.6 COPY - Copy Data

Description

Copies the 32-bit contents of a Register, Timer or Counter into another Register, Timer or Counter.
The contents of the first operand (source) is copied into the second (destination).

The PUTX, GETX and COPYX instructions are useful for the indexed transfer of data between Registers, Timers and Counters.
For PUTX the destination is indexed, for GETX the source is indexed, and for COPYX both the source and the destination are indexed.

Format

```
COPY[X]    [=] source (i)   ;source R T C
           [=] dest   (i)   ;destination R T C
```

Example

```
COPYX       R 10    ;move the contents of Register 10+Index
            R 50    ;to Register 50+Index
```

Flags

ACCU		Unchanged
Status Flags	E	Set if you copy a negative value to T or C, 0 is loaded
	P	Set according to the value copied
	Z	Set according to the value copied
	N	Set according to the value copied

See also

GET
PUT
MOV

Practical example

Move the contents of Registers 10..14 to Registers 50..54.

```
    SEI     K 0
Loop:
    COPYX   R 10
```

```
        R 50
INI     K 4
JR      H Loop
```

3.7 DEC - Decrement Register or Counter

Description
Decrements a Register or Counter by one.

This instruction is dependant on the ACCU state:
- Counters are only decremented if the ACCU is High.
- Registers are always decremented.

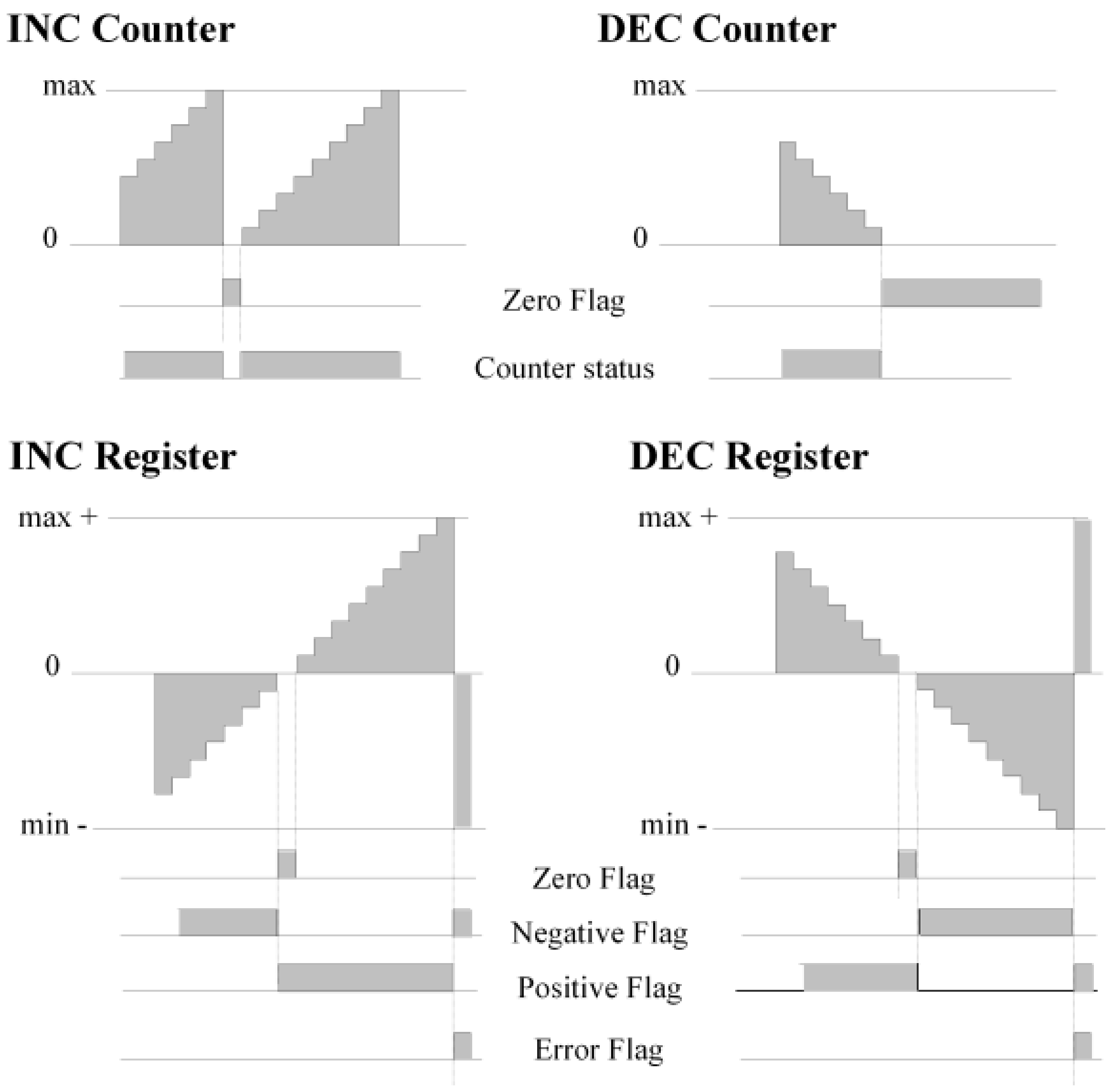
Format
```
DEC[X]   [=] element (i)   ;R or C to be decremented
```

Example
```
DEC       R 100    ;R 100 := R 100 - 1
```

Flags

ACCU		Unchanged For Timers and Counters, this instruction is executed only if the ACCU is High. For Registers this instruction is always executed
Status Flags	E	Set if underflow occurs
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
INC
SUB



3.8 DIGI - Digit In

Description
Moves Binary Coded Decimal (BCD) digits from Inputs, Outputs or Flags into a Register. A BCD digit is 4 bits (e.g. 4 Inputs), which represents a decimal digit (0..9).
The 1st operand is the number of digits to move (1..10).
The 2nd is the base Input, Output or Flag.
The 3rd is the destination Register.
The lowest addressed Input, Output or Flag becomes the least significant bit of the least significant digit in the destination Register.

Format
```
DIGI[X]    [=] digits      ;number of digits 1..10
           [=] source      ;source data I O F
           [=] dest (i)    ;destination R
```

Example
```
DIGI      2        ;read 3 BCD digits
          I 32     ;from Inputs 39..36 and 35..32
          R 100   ;into Register 100
```

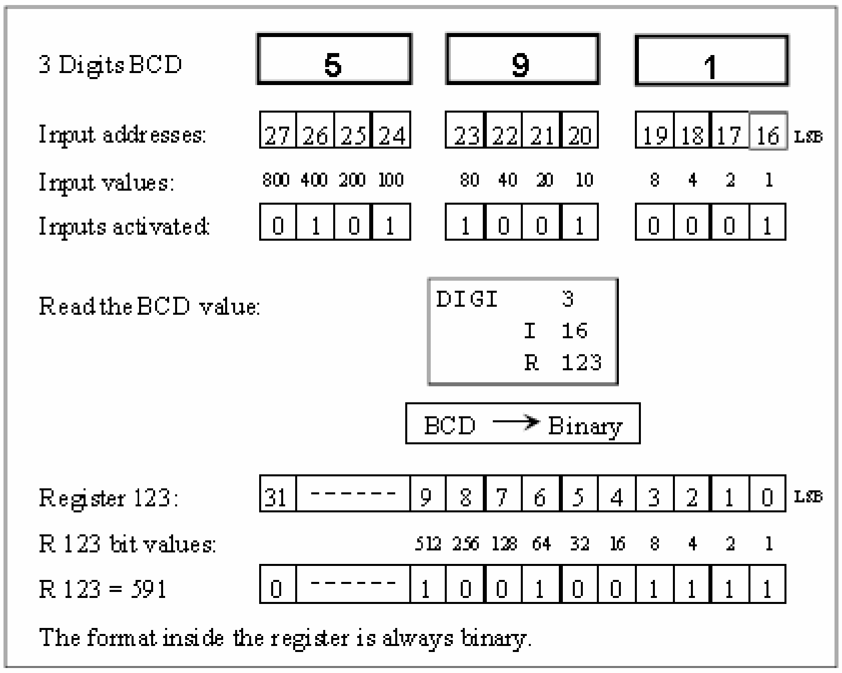
Flags

ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the value read
	Z	Set according to the value read
	N	Set according to the value read

See also

DIGIR
DIGO
DIGOR
BITI
BITIR

Practical example



3.9 DIGIR - Digit In Reversed

Description

Moves Binary Coded Decimal (BCD) digits from Inputs, Outputs or Flags into a Register. A BCD digit is 4 bits (e.g. 4 Inputs), which represents a decimal digit (0..9).

The 1st operand is the number of digits to move (1..10).
The 2nd is the base Input, Output or Flag.
The 3rd is the destination Register.

The lowest addressed Input, Output or Flag becomes the most significant bit of the most significant digit in the destination Register.

Format

```
DIGIR[X]    [=] digits      ;number of digits 1..10
            [=] source      ;source I O F
            [=] dest (i)    ;destination R
```

Example
```
DIGIR     2      ;read 2 BCD digits
          I 32   ;from Inputs 32..35 and 36..39
          R 100  ;into Register 100
```

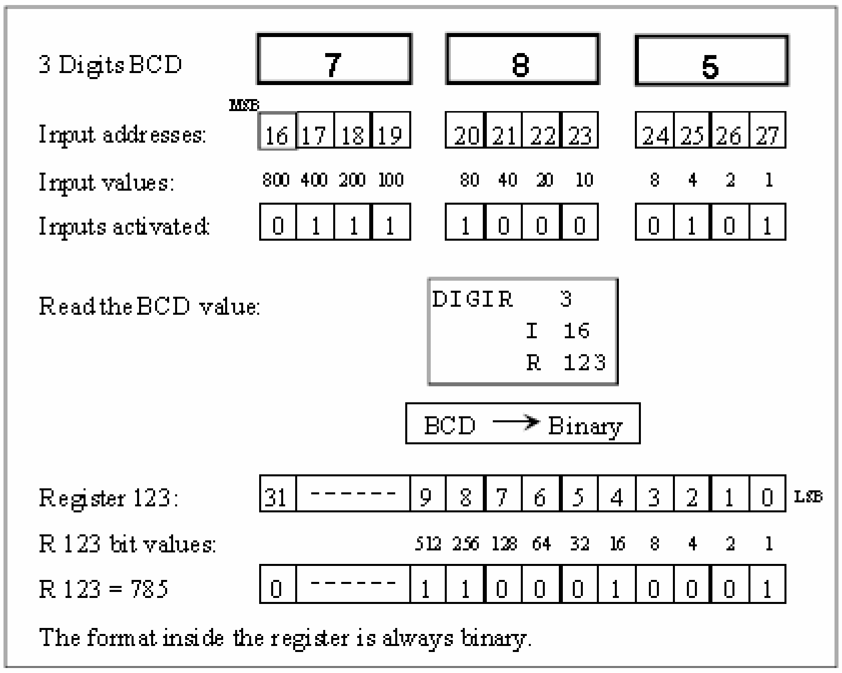
Flags

ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the value read
	Z	Set according to the value read
	N	Set according to the value read

See also

DIGI
DIGO
BITI
BITIR

Practical example



```
DIGIR    3
       I  16
       R  123
```

3.10 DIGO - Digit Out

Description

Moves BCD digits from a Register to a range of Outputs or Flags. A BCD digit consists of 4 binary bits.

The 1st operand is the number of BCD digits to move.
The 2nd is the source Register.
The 3rd is the base Output or Flag address.

The lowest addressed Output or Flag becomes the least significant bit of the least significant BCD digit.

Format
```
DIGO[X]   [=] digits      ;number of BCD digits 1..10
```

```
          [=] source (i)  ;source Register R
          [=] dest        ;destination O or F
```

Example

```
DIGO     2      ;write 2 BCD digits
         R 123  ;from Register 123
         O 40   ;to Outputs 47..44 and 43..40
```

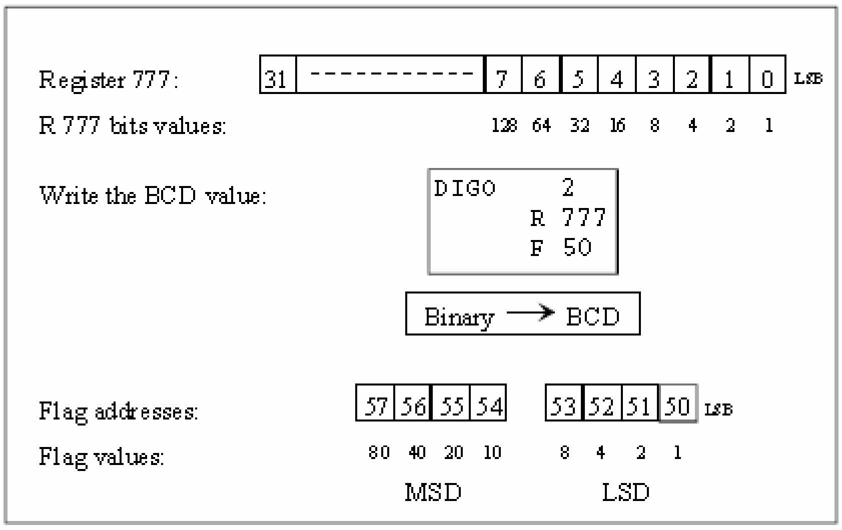
Flags

ACCU		Unchanged
Status Flags	E	Set if a BCD digit is invalid (> 9)
	P	Set according to the value read
	Z	Set according to the value read
	N	Set according to the value read

See also

DIGOR
DIGI
DIGIR
BITO
BITOR

Practical example



3.11 DIGOR - Digit Out Reversed

Description

Moves BCD digits from a Register to a range of Outputs or Flags. A BCD digit consists of 4 binary bits.

The 1st operand is the number of digits to move.
The 2nd is the source Register.
The 3rd is the base Output or Flag address.

The lowest addressed Output or Flag becomes the most significant bit of the most significant BCD digit.

Format

```
DIGOR[X]  [=] digits      ;number of BCD digits 1..10
          [=] source (i)  ;source Register R
          [=] dest         ;destination O or F
```

Example

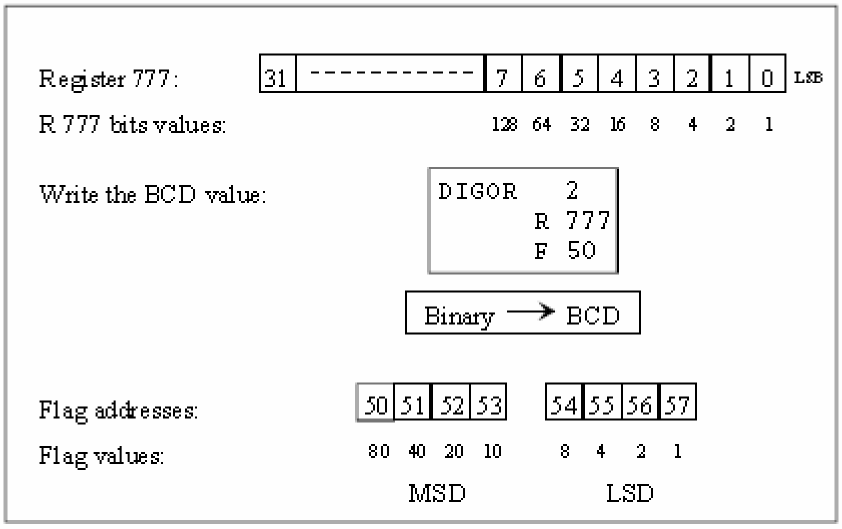
```
DIGOR     2        ;write 2 BCD digits
          R 123    ;from Register 123
          O 40     ;to Outputs 40..43 and 44..47
```

Flags

ACCU		Unchanged
Status Flags	E	Set if a BCD digit is invalid (> 9)
	P	Set according to the value read
	Z	Set according to the value read
	N	Set according to the value read

See also
DIGO
DIGI
DIGIR
BITOR
BITO

Practical example



3.12 DSP - Load Display Register

Description
The logical state of an Input, Output or Flag, or the contents of a Register, Timer, Counter or a constant, can be loaded into the Display Register.
The Display Register value can be displayed in decimal on the 7-Segment PCD2.F5xx display of a PCD1 or PCD2, and on the PCD8.P100 Programming Unit.
It can also be displayed by S-Bug's 'Display Display-register' command, or by entering DSP as the symbol name in the PG5's Watch Window.
It is useful as an error code or status display.

The operand cannot be supplied as a Function Block parameter.

Note
This instruction is not supported by new PCD types (NT systems, PCD3 and PCD2.M480 etc).

If not supported, XOB 8 (Invalid Opcode) is called. If XOB 8 is not present, the PCD will Halt.

Format

```
DSP     value    ;data to be displayed I O F T C R or K
```

Example

```
DSP     R 0      ;Display Register := contents of R 0
DSP     K 1234   ;Display Register := 1234
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

PCD1/2 Hardware Manuals.

3.13 EXOR - Exclusive-Or Registers

Description

The bits in the 1st Register is Exclusive-ORed with the bits in the 2nd Register, and the result is placed in the 3rd Register.
Exclusive-OR means that if either bit is a 1, bit not both bits, then result will be 1.

Format

```
EXOR[X]   [=] value1 (i)    ;R
          [=] value2        ;R
          [=] result (i)    ;R
```

Example

```
EXOR     R 1     ;Register 1 is exclusive-ORd
         R 2     ;with Register 2
         R 2     ;and the result is placed in Register 2
```

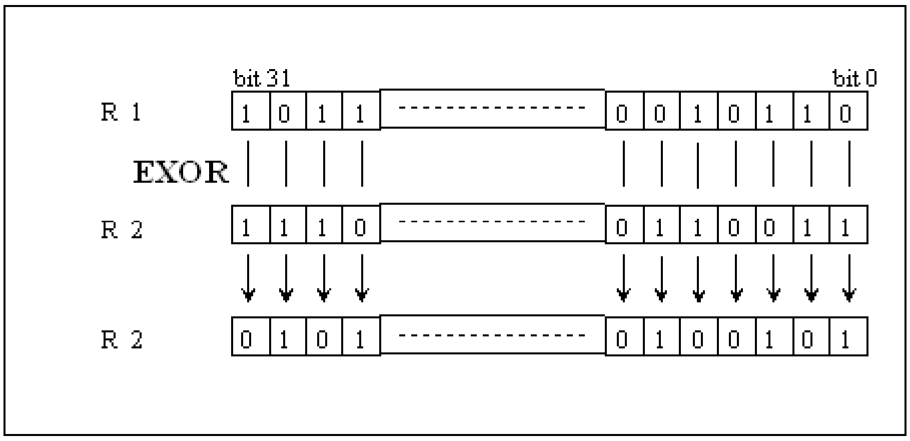
Flags

ACCU		Unchanged
Status Flags	E	Always set Low
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

OR

Practical example



3.14 GET - Get Data

Description

Copies the 32-bit contents of a Register, Timer or Counter into another Register, Timer or Counter.
It also allows the transfer of data from a Text or Data Block into a block of consecutive Registers, Timers or Counters.
The contents of the first operand (source) is copied into the second (destination). For `GETX`, the first operand (source) is indexed.

GET[X] will transfer a Text into a block of consecutive R/T/Cs, storing 4 characters per R/T/C, until the end of the Text is encountered (NUL terminator, 0).
If the Text is not a multiple of 4 characters long, unused bytes in the last R/T/C are unchanged.
Similarly, GET[X] can transfer 32-bit data items from a Data Block into a block of consecutive R/T/Cs, until the end of the DB.

If GET[X] tries to read from a Text or Data Block which doesn't exist then the Error flag is set and XOB 13 (Error Flag Set) is called.
If the indexed Text or Data Block number is out of range then XOB 12 is called (Index Register Overflow).

The PUTX, GETX and COPYX instructions are useful for the indexed transfer of data between Registers, Timers and Counters.
For PUTX the destination is indexed, for GETX the source is indexed, and for COPYX both the source and the destination are indexed.

Format

```
GET[X]   [=] source (i)    ;source R T C X or DB
         [=] dest          ;destination R T C
```

Example

```
GETX    R 10   ;move the contents of Register 10+Index
        R 50   ;to Register 50
```

Flags

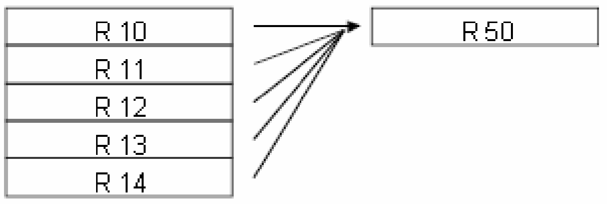
ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the value copied
	Z	Set according to the value copied
	N	Set according to the value copied

See also

PUT
COPY
MOV
Data Blocks (DB)
Texts (X)

Example 1
Move R 10 to R 50, then R 11 to R 50, up to R 14.

```
      SEI    K 0
LOOP: GETX   R 10
             R 50
      ...
      INI    K 4
      JR     H LOOP
```



Transfer between Text / Data Block and R/T/C
The instruction GET[X] can transfer from a Text into the R|T|C until the end of the Text (00, NUL terminator). If the Text does not end on an R|T|C boundary then the remainder of the R|T|C will be left unchanged.
Similarly, GET[X] can transfer the data items present in a Data Block to the R|T|C until the end of the Data Block.

A Data Block (DB) is a block which can hold large numbers of 32-bit values. Data Blocks can be used for storing values which are specific to a process to liberate R|T|C addresses for use by other processes.
If the instruction tries to read from a Text or Data Block which doesn't exist, then the Error flag is set and XOB 13 (Error Flag Set) is called.
If the indexed Text or Data Block number is greater than 8191 then XOB 12 is called (Index Register Overflow)

Example 2
Data Block declared in the source program:
```
DB 100 [5] 0h, 1h, 2h, 0a5a5a5a5h, 720h
```

Instruction to transfer the DB into a range of Registers:
```
GET    DB 100    ;transfer DB 100
       R 1000    ;into Registers 1000..1004
```

The result is:

Register	**Hex Value**
1000	00000000
1001	00000001
1002	00000002
1003	a5a5a5a5
1004	00000720

Example 3
Text declared in the source program:
```
TEXT 123 "THIS IS A TEXT 123"
```

Instruction to transfer Text 123 into registers 1000..1004:
```
GETX    X 123    ;transfer Text 123
        R 1000   ;into Registers 1000..1004
```

The result is:

Register	**Text Value**	**Hex Value**
1000	"THIS"	54484953
1001	" IS "	20495320
1002	"A TE"	41205445
1003	"XT 1"	58542031
1004	"23<0><0>"	32330000

3.15 INC - Increment Register or Counter

Description
Increment a Register or Counter value by 1.

This instruction is dependant on the ACCU:
- Counters are incremented only if the ACCU is High
- Registers are always incremented

Format
```
INC[X]   [=] element (i)   ;R or C
```

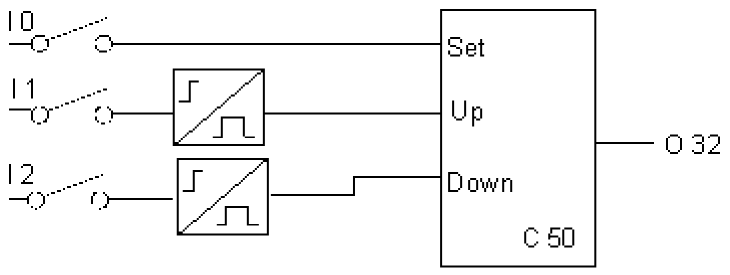
Example
```
INC    R 100     ;R 100 = R 100 + 1
```

Flags

ACCU		Unchanged For Timers and Counters, this instruction is executed only if the ACCU is High. For Registers this instruction is always executed
Status Flags	E	Set if overflow occurs
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

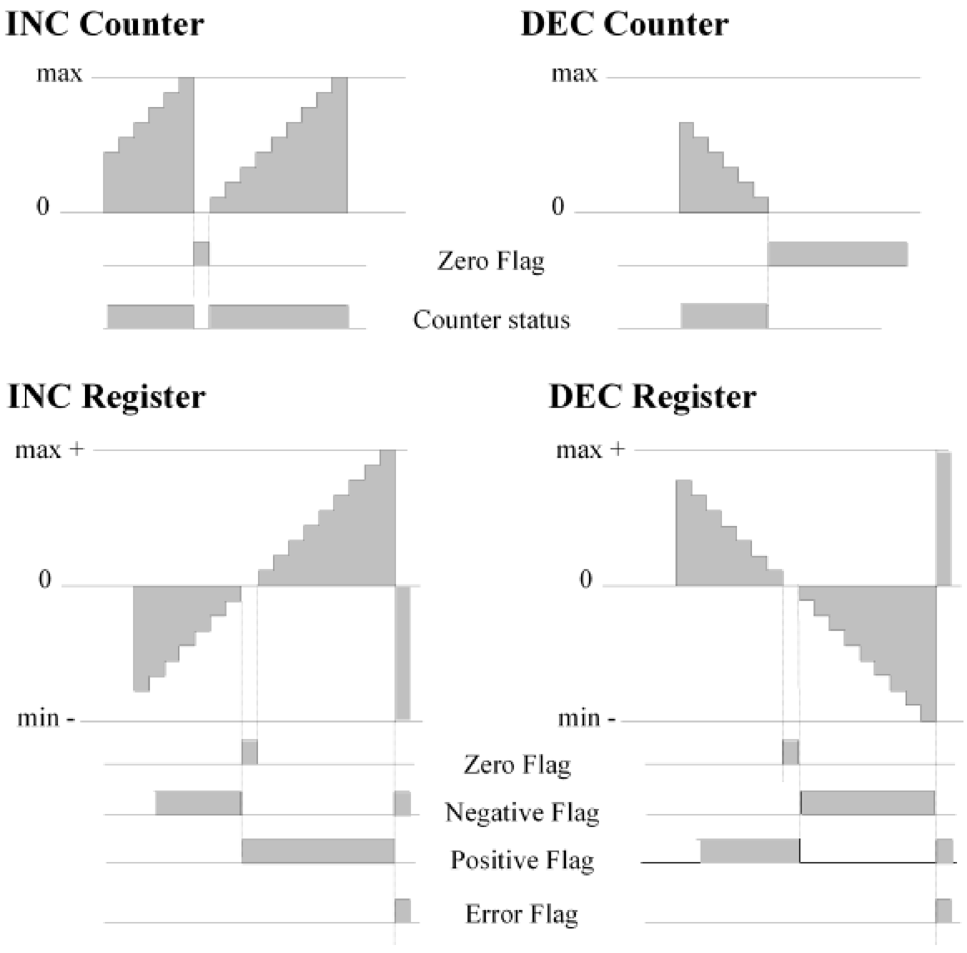
See also
DEC
ADD

Practical example
Up/down counter with pre-selection and display of the counter value.



```
    COB     0       ;start of COB
            0
    STH     I 0     ;if Input 0 is H
    LD      C 50    ;then load Counter 50 with 5
            5
                    ;else do nothing
```

```
STH    I 1     ;if Input 1 goes H
DYN    F 1     ;(edge detection)
INC    C 50    ;then increment Counter 50 by 1
               ;else do nothing
STH    I 2     ;if Input 2 goes H
DYN    F 2     ;(edge detection)
DEC    C 50    ;then decrement Counter 50 by 1
               ;else do nothing
STH    C 50    ;if counter 50 contents != 0
OUT    O 32    ;then set Output 32
               ;else reset Output 32
DSP    C 50    ;display Counter 50
ECOB
```



3.16 LD - Load 32-bit Value

Description
Load a Register, Timer or Counter with a 32-bit value.

For Timers and Counters:
- The instruction is only executed if the ACCU is High.
- Negative integer or floating point values are not supported (only Decimal, Hex, ANSI or Binary values).
- If a Timer is loaded, the Timer starts decrementing immediately according to the Timebase.
- The state of a Timer or Counter is High when it contains a non-zero value. Its state is Low when it contains zero.

For Registers:

- The operation is independent of the ACCU state, it is always loaded.
- The value can be a Decimal, Hex, ANSI, Floating Point or IEEE value.

Binary values are post-fixed with `Q` or `Y`, e.g. `1101Q`
Hex values are post-fixed with `H`, e.g. `0ABCDH`.
Floating point values must contain a decimal point or an exponent, e.g. `1.2`, `1E6`.
IEEE floats should be terminated by an `I`, e.g. `12.345I`
ANSI character values are enclosed in single quotes, e.g. `'a'`, `'A'`., `'abcd'`.

NOTES

- Because the value is 32 bits and each operand is 16 bits, this instruction uses three program lines - the 2nd and 3rd lines contain the 32-bit value as two 16-bit operands.
- CFB parameters are 16-bit values. The 32-bit LD instruction value cannot be supplied as a Function Block parameter. But you can use LDH and LDL to load a 16-bit Function Block parameter into the upper or lower 16 bits of a Register, Timer or Counter. Or pass the 32-bit value in a Register.
- LD T|C is only executed when the ACCU = H (1).
- LD R is always executed.

Format

```
LD[X]   [=] dest (i)    ;R T or C
            source      ;Decimal: -2147483648..+2147483647
                        ;Hex: 0H to FFFFFFFF
                        ;Binary: 0Y to 111...111Y (32 bits)
                        ;Floating point: ±5.42101E-20 to ±9.22337E+18
                        ;IEEE float: e.g. 1.23I
                        ;ANSI: 'A'-'Z', '0'-'9', '!', '?', 'abcd' etc.
```

If the source operand is an address, e.g. R 123 or a Register symbol, then the *address* of the source operand is loaded into the destination R T C, not the contents of the source address. See the example below. This allows addresses to be loaded into Registers for indirect addressing.

Example

```
LD      R 0          ;loads R 0 with floating point value 32.1
        3.21E1       ;(Timers & Counters must have +ve integer values)
LD      R 10         ;loads R 10 with the value 123,
        R 123        ;NOT the contents of R 123
LD      R 45         ;loads R 45 with the *value* of
        MySymbol     ;MySymbol, not the contents of Register MySymbol
```

Flags

ACCU	Unchanged For Timers and Counters, this instruction is executed only if the ACCU is High. For Registers this instruction is always executed
Status Flags	Unchanged

See also

LDH
LDL (16-bit loads)
Constants

3.17 LDH - Load High Word (upper 16 bits)

Description
Loads the upper 16 bits (31..16) of a Register, the lower 16 bits are not affected.
LDH cannot be used to load Timers or Counters, because the upper 16 bits cannot be loaded separately.

LDH and LDL (Load Low) allow 16-bit constants (0..65535) to be passed as Function Block parameters, or loaded directly.
A 32-bit value can be loaded using LDL and LDH, but LDL must be executed first because this sets the upper 16 bits to zero.

Values can be loaded in Decimal, Hex, ASCII or binary, but NOT floating point or IEEE which are 32-bit values.

Format
```
LDH[X]   [=] element (i)  ;R 0-4095*
         [=] value        ;Decimal: 0-65535
                          ;Hexadecimal: 0H-0FFFFH
                          ;Binary: 16 bits
```

Example
```
LDH      R 100        ;Loads bits 31-16 of Register 100
         0FFFFH       ;with FFFF Hex
                      ;R 100 = FFFFxxxx Hex
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
LDL
LD
Constants

Practical example
To load a Register in a Function Block with a 32-bit constant, you cannot use the LD instruction. Instead you must use LDL and LDH.
The upper and lower 16 bits of a constant can be separated using the Assembler statements '&' (AND) and '>>' (Shift Left).
In this example a constant (12345678) will be passed as parameter to a Function block where it is loaded into a Register. Remember that LDL must be done before LDH.

```
COB     0                    ;start COB
        0
CFB     0                    ;call Function Block 0
        12345678 & 0FFFFH    ;parameter 1 (lower 16 bits)
        12345678 >> 16       ;parameter 2 (upper 16 bits)
ECOB                         ;end of COB

FB      0                    ;start of FB 0
LDL     R 10                 ;load the lower 16 bits of Register 10
        = 1                  ;with the 1st parameter (lower 16 bits)
LDH     R 10                 ;load the upper 16 bits of Register 10
        = 2                  ;with the 2nd parameter (upper 16 bits)
...
```

```
EFB                          ;end of FB
```

3.18 LDL - Load Low Word (lower 16 bits)

Description

Loads the lower 16 bits (0..15) of a Register, Timer or Counter with a 16-bit value (0..65535); the upper 16 bits are always set to 0.
When values < 65535 are used, LDL can be used to load Counters, Timers or Registers instead of the 32-bit LD instruction.

This instruction is dependant on the ACCU:
- Timers and Counters are loaded only if the ACCU is High
- Registers are always loaded

LDL and LDH (Load High) allows 16-bit constants to be passed as Function Block parameters, or loaded directly.
LDH loads the upper 16 bits. A 32-bit value can be loaded using LDL and LDH together, but LDL must be executed first because this sets the upper 16 bits to zero.

Values can be loaded in Decimal, Hex, Binary or ANSI, but not Floating point or IEEE.

Format

```
LDL[X]    [=] element (i)  ;R T or C
          [=] value        ;Decimal: 0-65535
                           ;Hexadecimal: 0H-FFFFH
                           ;Binary: 16 bits, 0000000000000000Q ..
                           ;                 1111111111111111Q
```

Example

```
LDL    R 100     ;load Register 100 with FFFF Hex
       FFFFH     ;which is 65535 in decimal
                 ;R 100 = 0000FFFFH
```

Flags

ACCU	Unchanged For Timers and Counters, this instruction is executed only if the ACCU is High. For Registers this instruction is always executed
Status Flags	Unchanged

See also

LDH
LD
Constants

3.19 MOV - Move Data

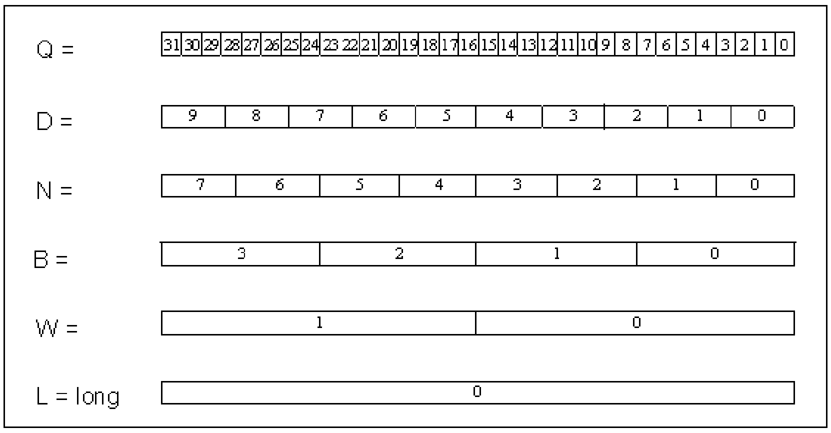
Description

Moves data from a Timer, Counter or Register into a Register. This is a 4-line instruction.
The 1st and 3rd operands are the source and destination.
The 2nd and 4th operands are the data type and position:

Q = Bit (moves 1 bit)	0..31
D = Digit (4 Bits BCD)	0..9
N = Nibble (4 Bits Binary)	0..7
B = Byte (8 Bits)	0..3
W = Word (16 Bits)	0..1

L = Long word (32 Bits) 0

The data types (Q, D etc.) of the 2nd and 4th operands must be the same, but source and destination positions may differ.



Format

```
MOV[X]     [=] source (i)       ;R T or C
               type position    ;Q|D|N|B|W|L see above
           [=] dest   (i)       ;R
               type position    ;Q|D|N|B|W|L see above
```

Practical example

Move the highest nibble (N 7) from Register 100 to the lowest nibble (N 0) of Register 101.

```
; Before: R100: 1111 1010 1010 1010 1010 1010 1010 1010
          R101: 0001 0001 0001 0001 0001 0001 0001 0001

; After:  R100: 1111 1010 1010 1010 1010 1010 1010 1010
          R101: 0001 0001 0001 0001 0001 0001 0001 1111
```

```
MOV     R 100
        N 7
        R 101
        N 0
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

COPY
GET
PUT
LD
LDH
LDL

3.20 NOT - Complement Register

Description

The contents of the 1st Register is inverted (1's complement) and stored in the 2nd Register.

Format

```
NOT[X]    [=] value  (i)   ;R
          [=] result (i)   ;R
```

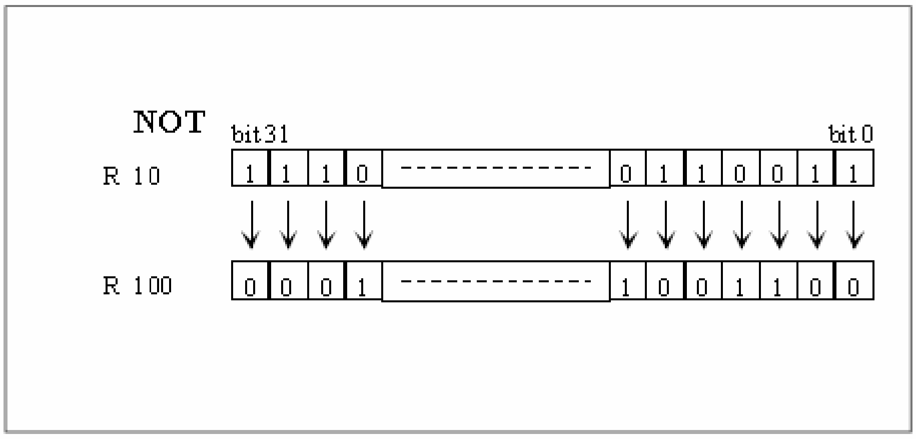
Example

```
NOT     R 10    ;invert the contents of Register 10
        R 100   ;and put the result in Register 100
```

Flags

ACCU		Unchanged
Status Flags	E	Always set Low
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

Practical example



3.21 OR - Or Registers

Description

The contents of the 1st Register is logically ORed with the contents of the 2nd Register, and the result is placed in the 3rd Register.

Format

```
OR[X]   [=] value1 (i)   ;R
        [=] value2       ;R
        [=] result (i )  ;R
```

Example

```
OR     R 1    ;ORs Register 1
       R 2    ;with Register 2
       R 3    ;and puts the result in the Register 3
```

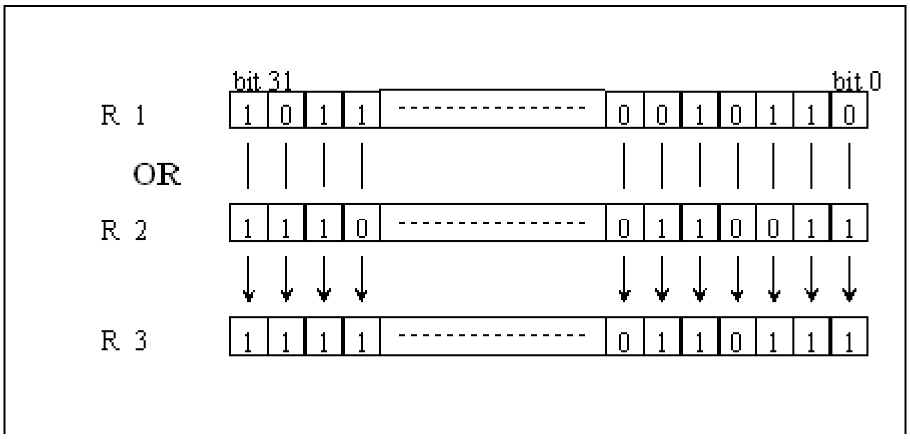
Flags

ACCU		Unchanged
Status Flags	E	Always set Low
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

EXOR

Practical example



3.22 PUT - Put Data

Description
Copies the 32-bit contents of a Register, Timer or Counter into another Register, Timer or Counter.
It also allows the transfer of data from a block of consecutive Registers, Timers or Counters into a Text or Data Block.
The contents of the first operand (source) is copied into the second (destination).

PUT[X] can transfer a block of consecutive R|T|Cs into a Text, until the end of the Text is encountered (NUL terminator, 0).
If there is a NUL (00) character in an R|T|C then it is changed into a space (20H).
Similarly, GET[X] can transfer 32-bit data items from a block of consecutive R|T|Cs into a Data Block until the end of the DB.

If PUT[X] tries to write to a Text or Data Block which doesn't exist then the Error flags is set and XOB 13 (Error Flag Set) is called.
If the indexed Text or Data Block number is out of range then XOB 12 is called (Index Register Overflow).

The PUTX, GETX and COPYX instructions are useful for the indexed transfer of data between Registers, Timers and Counters.
For PUTX the destination is indexed, for GETX the source is indexed, and for COPYX both the source and the destination are indexed.

Format
```
PUT[X]    [=] source     ;source R T C
          [=] dest (i)   ;destination R T C, X or DB
```

Example
```
PUTX     R 10    ;move the contents of Register 10
         R 50    ;into Register 50 + Index
```

Flags

ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the value copied
	Z	Set according to the value copied

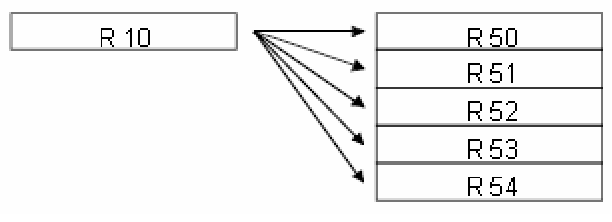
N Set according to the value copied

See also
GET
COPY
MOV

Notes
PUT cannot change the Text or Data Block length, and it cannot write beyond the end of the Text or DB.
PUT cannot transfer values into a Text or to a Data block if EPROM or Flash memory is used, or RAM with the jumper in the "Write Protect" (WP) position.

Example 1
Move the contents of Register 10 into Registers 50 to 54.

```
       SEI    K 0
LOOP:  PUTX   R 10
              R 50
       INI    K 4
       JR     H Loop
```



Example 2
Data Block as declared in the source program:
```
DB 100 [5]    ;Initial values are zero
```

Contents of Registers:

Register	Decimal Value
1000	00000001
1001	00000002
1002	00000003
1003	01234567
1004	00000720

Instruction:
```
PUT    R 1000   ;transfer Registers 1000..1004
       DB 100   ;into Data Block 100
```

Result as displayed with the debugger in decimal:
```
DB 100 [0]:      1      2      3     1234567    720
```

Example 3
Text as declared in the source program:
```
TEXT 100 [17]    ;text containing 17 spaces
```

Contents of Registers:

Register	ANSI Value	Hex Value
1000	"THIS"	54484953
1001	" IS "	20495320
1002	"A TE"	41205445
1003	"XT 1"	58542031
1004	"23 "	32332020

Instruction:

```
PUT     R 1000   ;transfer Registers 1000..1004
        X 100    ;into Text 100
```

Result as displayed with the debugger.T
he size of the Text is unchanged at 17 characters, so the last three characters from Register 1004 are not transferred.
```
TEXT 100 "THIS IS TEXT 12"
```

3.23 ROTD - Rotate Registers Down

Description
Rotates the contents of a block of Registers down one place.
The 1st and 2nd operands indicate the start and end of the block of Registers to be rotated.
After the rotate, the highest Register contains the value of the lowest.
Either the higher or the lower Register can be specified first.

Format
```
ROTD   [=] start    ;R
       [=] end      ;R
```

Example
```
ROTD   R 100   ;rotate R 100 to R 105 down one address
       R 105   ;R 100=R 101 .. R 104=R 105, R 105=R 100
```

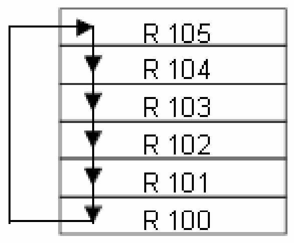
Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
ROTU
SHIU
SHID

Practical example

```
ROTD  R 100
      R 105
```



3.24 ROTL - Rotate Register Left

Description
The contents of the addressed Register is rotated left by the number of bits given in the 2nd operand.
The most significant bit 31 is copied into the least significant bit 0.
The ACCU is set to state of the last bit that was rotated.

Format
```
ROTL[X]   [=] reg (i)   ;R
```

```
          [=] nbits     ;number of bits to rotate 1..32
```

Example

```
ROTL    R 10   ;rotate Register 10 left
        4      ;by 4 bits
```

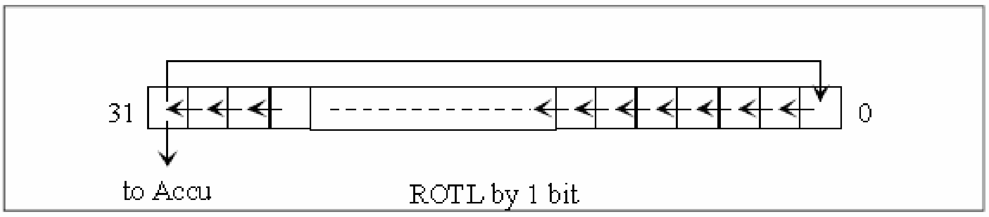
Flags

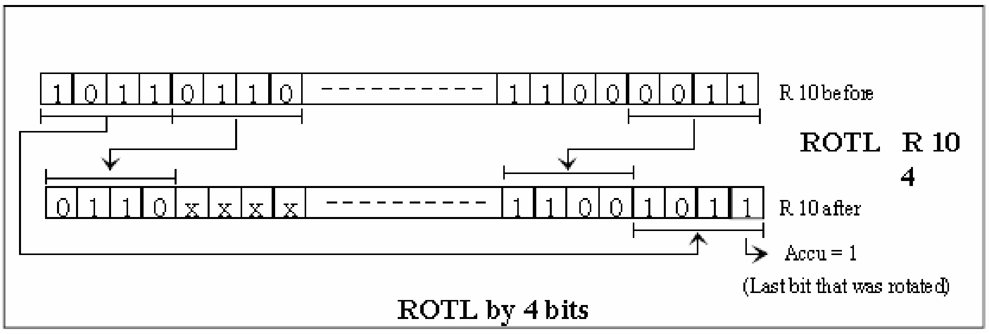
ACCU	Set to the state of the last bit which was rotated
Status Flags	Unchanged

See also

ROTR
SHIL
SHIR

Practical example





3.25 ROTR - Rotate Register Right

Description

The contents of the addressed Register is rotated right by the number of bits given in the 2nd operand.
The least significant bit 0 is copied into the most significant bit 31.
The ACCU is set to state of the last bit that was rotated.

Format

```
ROTR[X]   [=] reg (i)   ;R
          [=] nbits     ;number of bits to rotate 1..32
```

Example

```
ROTR     R 10   ;rotate Register 10 right
         4      ;by 4 bits
```

Flags

ACCU	Set to the state of the last bit which was rotated
Status Flags	Unchanged

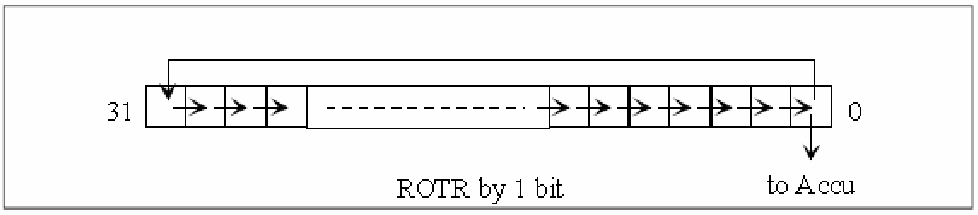
See also

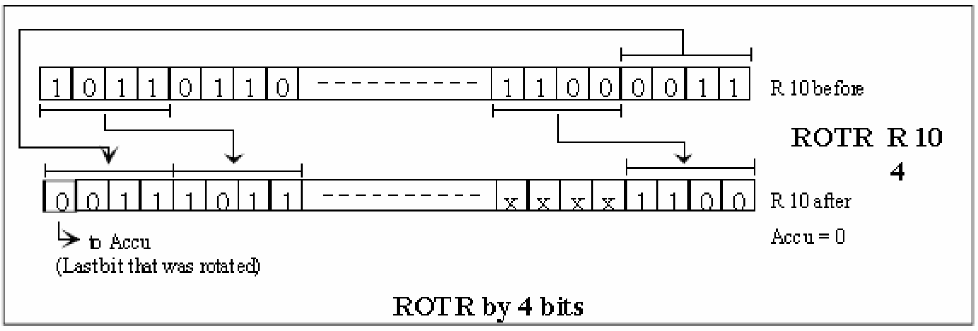
[ROTL]

[SHIL]

[SHIR]

Practical example





3.26 ROTU - Rotate Registers Up

Description

Rotates the contents of a block of Registers up one place.
The 1st and 2nd operands indicate the start and end of the block of Registers to be rotated.
After the rotate, the lowest Register contains the value of the highest.
Either the higher or the lower Register can be specified first.

Format

```
ROTU    [=] start    ;R
        [=] end      ;R
```

Example

```
ROTU     R 100    ;rotate R 100 to R 105 up one place
         R 105    ;R 100=R 105, R 101=R100 .. R 105=R 104
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

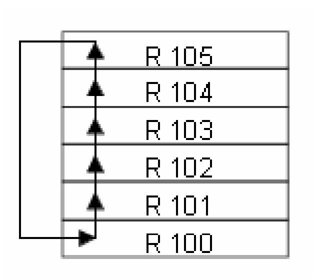
[ROTD]

[SHIU]

SHID

Practical example

```
ROTU   R 100
       R 105
```



3.27 SHID - Shift Registers Down

Description
Shifts the contents of a block of Registers down one place.
The 1st and 2nd operands are the start and end of the block of Register to be shifted.
After the shift, the highest Register contains zero, and the lowest overwrites the Register below.
Either the upper or the lower Register can be specified first

Format
```
SHID     [=] start    ;R
         [=] end      ;R
```

Example
```
SHID     R 100    ;shift R 100 to R 105 down one place
         R 105    ;R 99=R 100 .. R 104=R 105, R 105=0
```

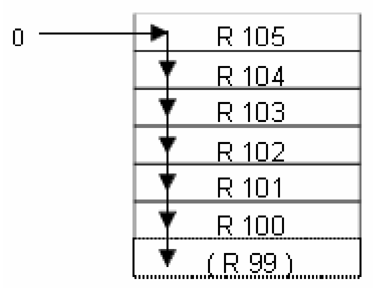
Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
SHIU
ROTU
ROTD

Practical example

```
SHID   R 100
       R 105
```



Note
This instruction overwrites one Register more than those specified: the Register which precedes the start of the block is overwritten.

3.28 SHIL - Shift Register Left

Description

The contents of the addressed Register is shifted left by the number of bits given by the second operand.
The content of the ACCU (1 or 0) is shifted in from bit 0 (the least significant bit), n times.
At the end of the operation, the ACCU is set to the state of the last bit shifted out of the Register.

Format

```
SHIL[X]    [=] reg (i)    ;R
           [=] nbits      ;number of bits to shift 1..32
```

Example

```
SHIL      R 10  ;shift Register 10 left
          4     ;by 4 bits (divide by 16)
```

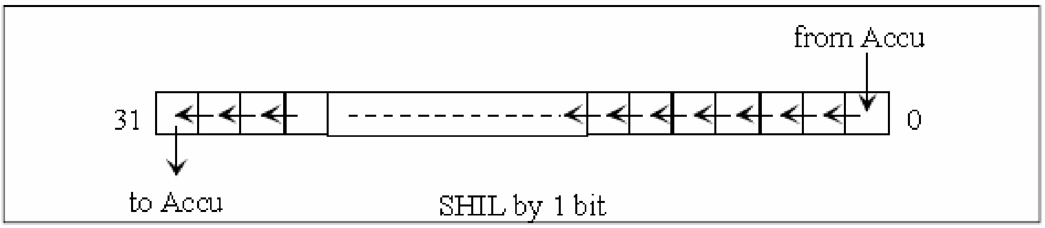
Flags

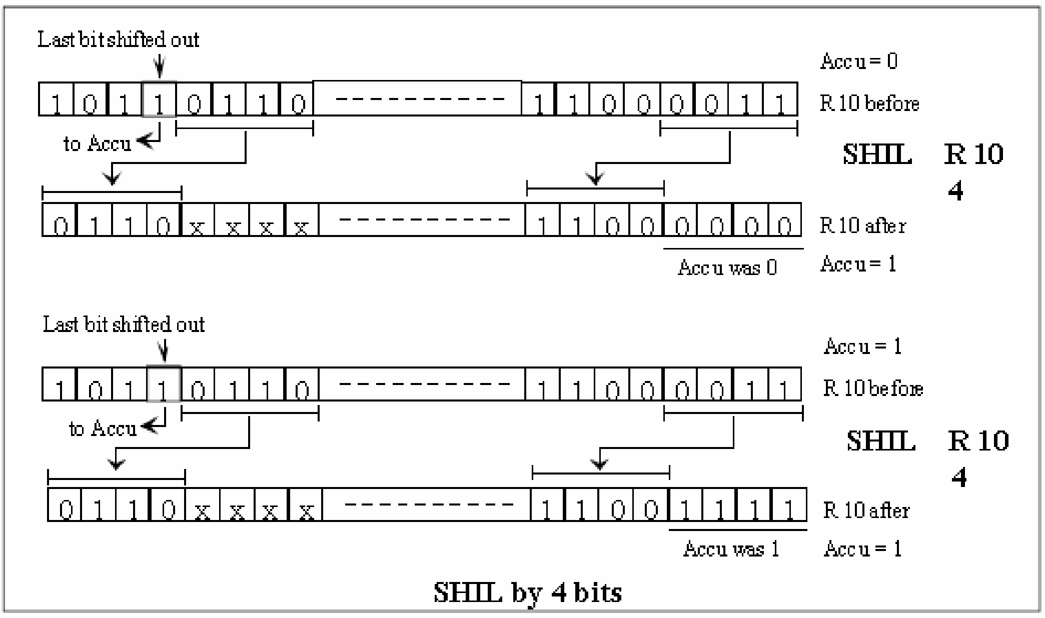
ACCU	Set to the state of the last bit shifted out of the Register
Status Flags	Unchanged

See also

SHIR
ROTL
ROTR

Practical example





3.29 SHIR - Shift Register Right

Description
The contents of the addressed Register is shifted right by the number of bits given by the second operand.
The contents of the ACCU (1 or 0) is shifted in from bit 31 (the most significant bit), n times.
At the end of the operation, the ACCU is set to the state of the last bit shifted out of the Register.

Format
```
SHIR[X]   [=]  reg (i)   ;R
          [=]  bits     ;number of bits to shift 1..32
```

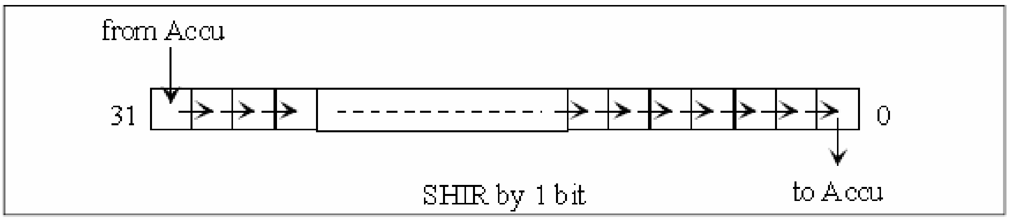
Example
```
SHIR     R 10  ;shift Register 10 right
         16    ;by 16 bits (divide by 65536)
```

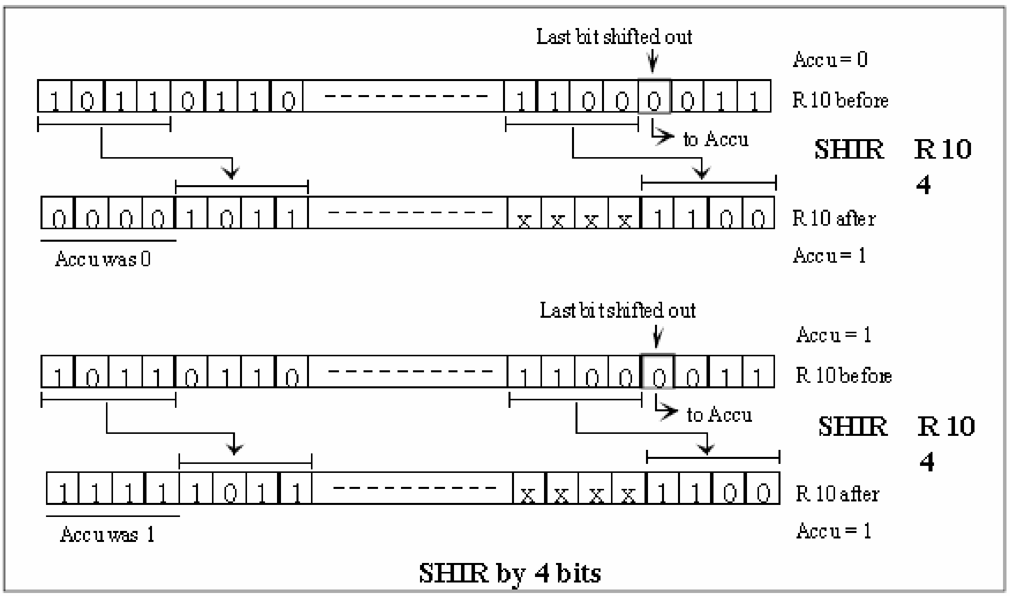
Flags

ACCU	Set to the state of the last bit shifted out of the Register
Status Flags	Unchanged

See also
SHIL
ROTL
ROTR

Practical example





3.30 SHIU - Shift Registers Up

Description
Shifts the contents of a block of Registers up one place.
The 1st and 2nd operands are the start and end of the block of Registers to be shifted.
After the shift, the lowest Register contains zero, and the highest overwrites the Register above.
Either the upper or the lower Register can be specified first.

Format
```
SHIU    [=] start    ;R
        [=] end      ;R
```

Example
```
SHIU    R 100    ;shift R 100 to R 105 up one place
        R 105    ;R 100=0, R 101=R 100 .. R 106=R 105
```

Flags

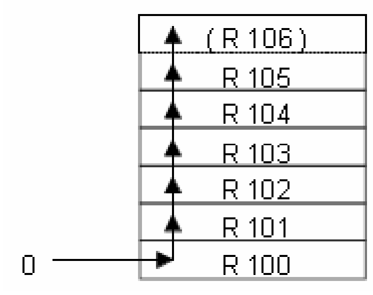
ACCU	Unchanged
Status Flags	Unchanged

See also
SHID
ROTU
ROTD

Note
This instruction overwrites one Register more than those specified: the Register which follows the end of the block is also overwritten.

Practical example

```
SHIU   R 100
       R 105
```



3.31 TFR - Transfer Data

Description
This instruction enables the indexed data transfer of individual values from a Data Block or a Text into Registers, Timers or Counters; and vice versa.

Format
To copy an individual 32-bit value from a Data Block or Text into a Register, Timer or Counter:
The 1st operand is the Data Block or Text containing the value to transfer.
The 2nd operand is the position of the value inside the Data Block or Text, which can be given as a constant or indirectly via a Register.
The 3rd operand is the destination Register, Timer or Counter.

```
TFR[X]   [=] source     ;DB X
         [=] position   ;R K
         [=] dest (i)   ;R T C
```

To copy a Register, Timer or Counter into a Data Block or Text:
The 1st operand is the Register, Timer or Counter containing the value to transfer (source).
The 2nd operand is the destination Data Block or Text.
The 3rd operand is the position inside the Data Block or Text where the value is to be transferred, this position can be given as a constant or indirectly via a Register.

```
TFR[X]   [=] source (i)  ;R T C
         [=] dest        ;DB X
         [=] position    ;R K
```

Example

```
TFR     DB 4010    ;copy from the Data Block 4010
        K 13       ;the value at position 13
        R 26       ;to Register 26

TFR     R 120      ;copy Register 120
        DB 4025    ;to Data Block 4025
        K 6        ;at position 6
```

Flags

ACCU	Unchanged

Status Flags	E	Set if position is beyond the end of the DB
	P	Set according to the value copied
	Z	Set according to the value copied
	N	Set according to the value copied

See also

PUT

GET

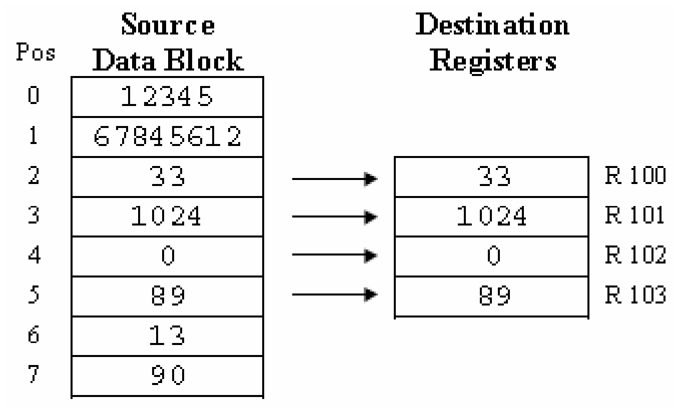
Note

Access to DBs 4000..8191 in Extension Memory is significantly faster than to DBs 0..3999.
It is therefore recommended that this instruction should be used mainly with DBs 4000..8191.
Data Blocks 0..3999 in Text/DB memory can hold up to 383 values (0..382). Data Blocks in Extension (Data) Memory can hold up to 16384 values (0..16383).

Practical examples

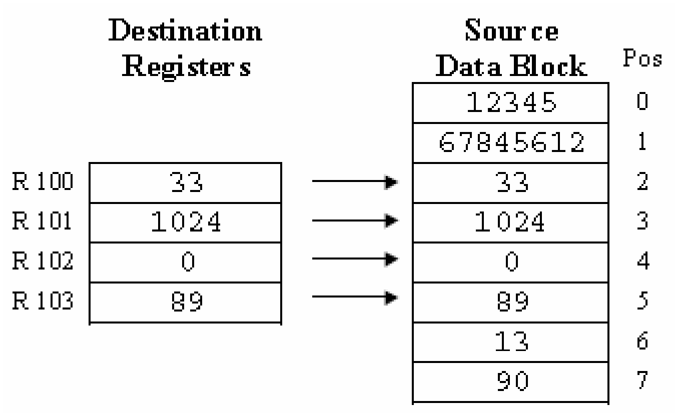
From Data Block 4010, the 4 values from positions 2..5 are copied to Registers 100..103.

```
      LD     R 999
             2
      SEI    K 0
LOOP: TFRX   DB 4010
             R 999
             R 100
      INC    R 999
      INI    K 3
      JR     H LOOP
      ....
```



Registers 100..103 are copied to positions 2..5 of Data Block 4010:

```
      LD     R 999
             2
      SEI    K 0
LOOP: TFRX   R 100
             DB 4010
             R 999
      INC    R 3
      JR     H LOOP
      ...
```



3.32 TFRI - Transfer Data Indirect

Description

Transfers a singe Register, Timer or Counter value to or from a Data Block or Text using Register-indirect addressing. The source and destination media addresses are supplied in Registers.

Notes

- For firmware versions earlier than 1.20.00, the max. Register address for indirect instructions is 8191.
- To use Register addresses 8192..16383 with firmware version 1.2.00 or later, set the Build Option "Use 16-bit Register and Flag addressing" to Yes.
- This instruction cannot be used with Function Block parameters (= n).

Format

To copy an individual 32-bit value from a Data Block or Text into a Register, Timer or Counter:
The 1st operand defines the source type Data Block or a Text (DB/X) followed by Reg1 which is the number of a Register containing the DB or Text number.
The 2nd operand is the position of the value inside the Data Block or Text, which can be given as a constant or indirectly via a Register.
The 3rd operand defines the destination type (R/T/C) followed by Reg2 which is the number of a Register containing the destination Register, Timer or Counter number.

```
TFRI     type  source    ;DB|X  Reg1
         position        ;R K
         type  dest      ;R|T|C Reg2
```

To copy a Register, Timer or Counter into a Data Block or Text:
The 1st operand defines source element type (R/T/C) followed by `Reg1` which is the number of a Register containing the Register, Timer or Counter number.
The 2nd operand defines the destination type (DB/X) followed by `Reg2` which is the number of a Register containing the destination DB or Text number.
The 3rd operand is the position inside the Data Block or Text where the value is to be transferred, this position can be given as a constant or indirectly via a Register.

```
TFRI     type  source    ;R|T|C Reg1
         type  dest      ;DB|X  Reg2
         position        ;R|K
```

Examples

Transfer the element at position 10 of Data Block 4000 to Register 4095:

```
LD     R 100     ;load the DB number
       4000
LD     R 101     ;load the Register number
       4095
TFRI   DB 100    ;transfer: DB=source type, 100=req with DB number
       K 10      ;DB position 10
       R 101     ;R=dest type, 101=reg with actual Register number
```

Transfer the value if Counter 1000 to position 50 of Data Block 4000:

```
LD     R 100     ;load the DB number
       4000
LD     R 101     ;load the position
       50
LD     R 102     ;load the Counter number
       1000
TFRI   C 102     ;transfer: C=source type, 102=reg with Counter number
       DB 100    ;destination DB
       R 101     ;R=dest type, 101=reg with the DB position
```

Flags

ACCU	Unchanged

Status Flags	E	Unchanged
	P	Set according to the value copied
	Z	Set according to the value copied
	N	Set according to the value copied

See also

TFR
PUT
GET

Notes

- On old PCD models, access to DBs 4000..8191 in Extension Memory (Data Memory) is significantly faster than to DBs 0..3999. It is therefore recommended that this instruction should be used mainly with DBs 4000..8191.
- Data Blocks 0..3999 in Text/DB memory can hold up to 383 values (0..382). Data Blocks in Extension Memory can hold up to 16384 values (0..16383).

4 Index Register Instructions

It is frequently necessary for series of Inputs, Outputs, Flags etc. to be dealt with in the same way (for example resetting of non-volatile Flags or Registers).
In cases like this, long programs can be drastically shortened with the help of address indexing.

Each COB or XOB has its own Index Register. This register is used for indexed addressing, where the contents of the Index Register is added to the operand value to provide the actual address.

Indexing instructions are always ended with an 'X', for Example STHX, BITIX.
The Index Register can be loaded or saved, incremented up to a given limit, or decremented down to a given limit.

SEI	Set Index register
INI	Increment Index register
DEI	Decrement Index register
STI	Store Index register
RSI	Restore Index register

4.1 SEI - Set Index Register

Description
The current Index Register is loaded with the supplied constant (K 0-8191) or the contents of the indicated Register.
Each COB has its own Index register, and all XOBs share their own Index Register.

The range for the Index Register value is 0..8191 (13 bits).

If a value > 8191 is loaded, the Index Register is set to 8191 and XOB 12 is called.
If a value < 0 is loaded, the Index register is set to 0 and XOB 12 is called.

Format
```
SEI   [=] value   ;K 0..8191, R
```

Example
```
SEI   K 32   ;loads Index Register with the value 32
SEI   R 32   ;loads Index Register with the contents of Register 32
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
INI
DEI
STI
RSI

Practical example
The state of the Input whose address is given by a BCD encoder switch must be transferred to Output 32.

```
    COB    0
           0
    DIGI    2          ;read 2 BCD digits
            I 24       ;from Inputs 24..31
            R 500      ;and store them in Register 500
    SEI     R 500      ;load Index with the contents of Register 500
    STHX    I 0        ;if Input (0 + Index) is High
    OUT     O 32       ;then set Output 32
                       ;else reset Output 32
    ECOB
```

4.2 INI - Increment Index Register

Description

The current Index Register is compared to the value of the operand (supplied K constant or the contents of a Register).
If the Index Register is less than this value, the Index Register is incremented and the ACCU is set High (1).
If the Index Register is equal or greater than the value of the operand, the Index Register is NOT incremented and the ACCU is set Low (0).

If the value in the operand is greater than 8191 (old systems) or greater than 16383 (NT systems), or less than 0 then the Index Register is not modified, XOB 12 is called (if programmed) and the ACCU is set Low. This can happen if the operand is a Register which contains an out-of-range Index Register value.

Each COB has its own Index register, and all XOBs share their own Index Register.

Format

```
INI    [=] value   ;K or R
```

Example

```
INI    K 100    ;increment Index register if < 100
INI    R 333    ;increment Index register if lower than the
                ; contents of R 333
```

Flags

ACCU	Set Low if the Index Register is greater than or equal to the operand value. Set High if the Index Register is less than the value of the operand. Set Low if the operand is out of range, and XOB 12 is called.
Status Flags	Unchanged

See also

DEI
SEI

Practical example

At start-up, Registers 1500 to 1999 must be reset (value 0).

```
    XOB     16          ;XOB executed at start-up
    ...
    SEI     K 0         ;set Index Register to 0
Repeat:
    LDX     R 1500      ;load Register (1500 + Index Reg.)
            0           ;with 0
    INI     K 499       ;increment Index Register by 1
    JR      H Repeat    ;until Index Register > 499
   (ACC H               ;in case subsequent code needs it)
    ...
    EXOB
```

4.3 DEI - Decrement Index Register

Description
The current Index Register is compared to the value of the operand (K constant or the contents of a Register).
If the Index Register is greater than this value, the Index Register is decremented and the ACCU is set High (1).
If the Index Register is equal or less than the value of in the operand (constant or Register contents), the Index register is NOT decremented and the ACCU is set Low (0).

If the value in the operand is greater than 8191 (old systems) or greater than 16383 (NT systems), or less than 0 then the Index Register is not modified, XOB 12 is called (if programmed) and the ACCU is set Low. This can happen if the operand is a Register which contains an out-of-range Index Register value.

Each COB has its own Index register, and all XOBs share their own Index Register.

Format
```
DEI   [=] value   ;K 0..8191, R
```

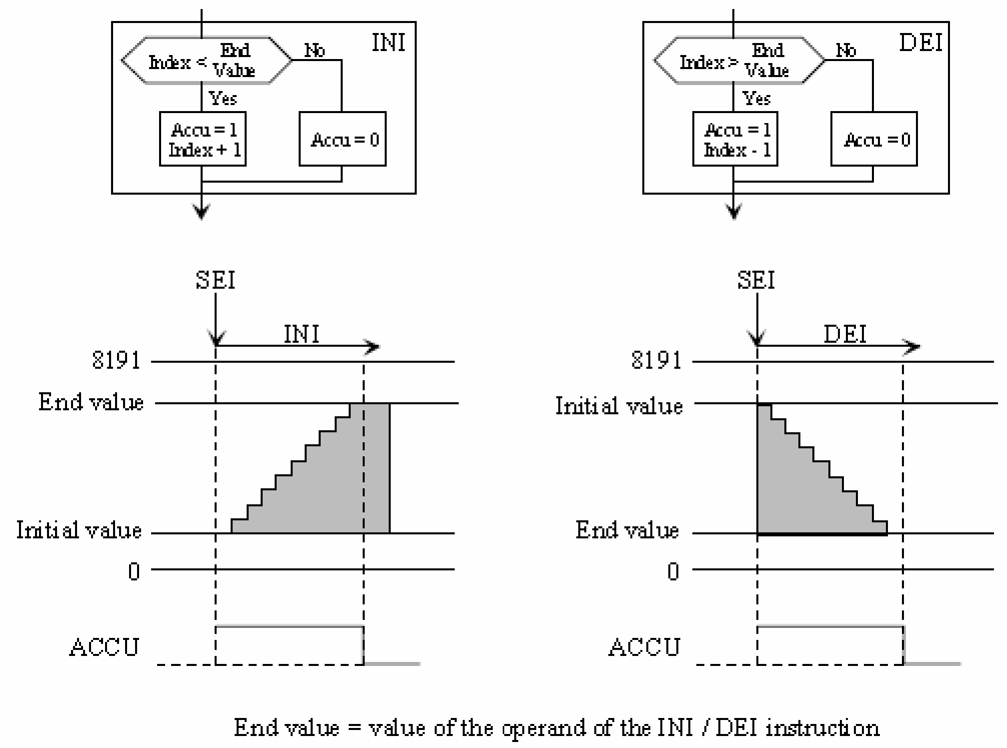
Example
```
DEI    K 100    ;decrements Index Register if > 100
DEI    R 444    ;decrements Index Register if greater that
                ;than the contents of Register 444
```

Flags

ACCU	Set Low if the Index Register is less than or equal to the operand value. Set High if the Index Register is greater than the value of the operand. Set Low if the operand is out of range, and XOB 12 is called.
Status Flags	Unchanged

See also
INI
SEI

Practical example



4.4 STI - Store Index Register

Description
The value in the current Index Register is stored in the given Register.
It can be re-loaded into the Index Register using the RSI instruction.
The Index Register is unchanged.

Format
```
STI     [=] dest    ;destination R
```

Example
```
STI     R 100       ;stores the Index Register value in Register 100
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
RSI

4.5 RSI - Restore Index Register

Description
Loads the Index Register with the contents of the given Register.
The value in the Register will typically be an Index Register value saved by the STI instruction.

If the value to be restored is less than 0 or greater than 8191 (old systems) or greater than 16383 (NT systems), then XOB 12 is called (if present), and the Index Register is set to the minimum value (0) or maximum value (8191 or 16383).

Format

```
RSI    [=] source   ;R
```

Example

```
RSI    R 100    ;load the Index register with the contents of
                ;Register 100 (same as: SEI  R 100)
...
LD     R 100
       -1
SEI    R 100    ;Index Register set to 0, XOB 12 is called
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

STI

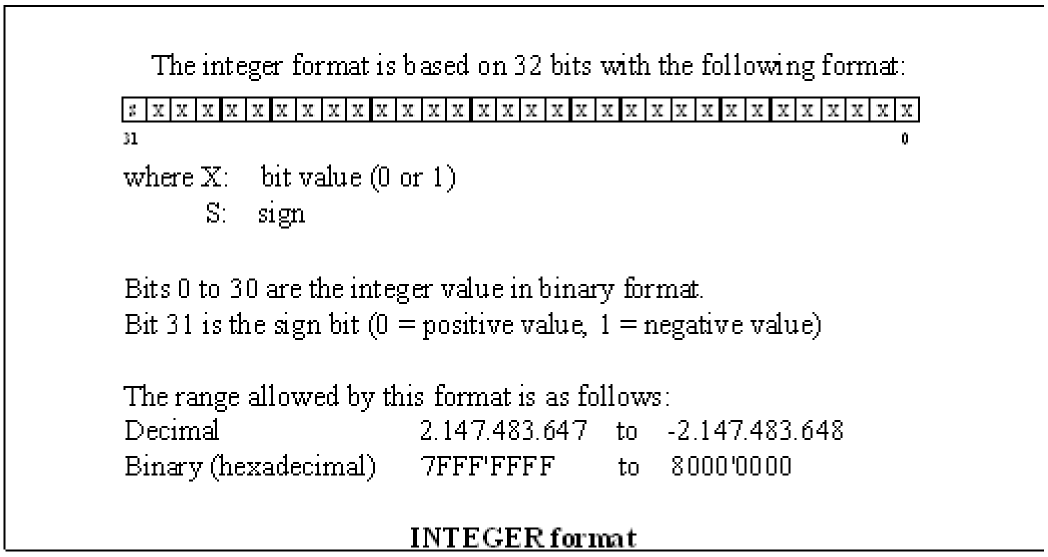
SEI

5 Integer Instructions

The integer arithmetic instructions work with Registers containing signed 32-bit values with the range:
-2'147'483'648 to +2'147'483'647 (080000000H to 07FFFFFFFH)
(except UDIV and UMUL which support 32-biit unsigned values)

ADD	Add Registers
SUB	Subtract Registers
MUL	Multiply Registers
DIV	Divide Registers
SQR	Square Root
CMP	Compare Registers
UMUL	Unsigned Multiply Registers
UDIV	Unsigned Divide Register

For floating point values, the Floating Point instructions must be used.



The integer format is based on 32 bits with the following format:

S X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
31 0

where X: bit value (0 or 1)
S: sign

Bits 0 to 30 are the integer value in binary format.
Bit 31 is the sign bit (0 = positive value, 1 = negative value)

The range allowed by this format is as follows:

Decimal	2.147.483.647	to	-2.147.483.648
Binary (hexadecimal)	7FFF'FFFF	to	8000'0000

INTEGER format

5.1 ADD - Add Registers

Description
Signed integer addition.
Adds the contents of the 1st Register or constant to the contents of the 2nd Register or constant, and stores the result in the 3rd Register.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format
```
ADD[X]   [=] value1 (i)    ;R K
         [=] value2        ;R K
         [=] result (i)    ;R
```

Example
```
ADD    R 20       ;add 123 to Register 20
       K 123      ;(range is K 0..16383)
       R 20
```

Tip: To add a value larger than K 16383, first load a Register with the value, then add the Registers.

Flags

ACCU		Unchanged
Status Flags	E	Set on overflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

FADD

Practical example

Read two numbers, add them and put the result in another Register.
The two numbers come from BCD encoders (2 digits) on Inputs 16 to 23, and 24 to 31.

```
    COB     0
            0
    DIGI    2        ;read 2 digit
            I 16     ;from Inputs 16..23
            R 100    ;and store them in R 100
    DIGI    2        ;read 2 digits
            I 24     ;from Inputs 24..31
            R 200    ;and store them in R 200
    ADD     R 100    ;R 0 = R 100 + R 200
            R 200
            R 0
    ...
    ECOB
```

5.2 CMP - Compare Registers

Description

Compares the contents of the 1st Register or constant with the contents of the 2nd Register or constant.
This is done by subtracting the 2nd value from the 1st value, the Status flags are set according to the result.
The contents of the Registers are unchanged.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format

```
CMP[X]   [=] value1 (i)    ;R K
         [=] value2        ;R K
```

Example

```
CMP     R 0      ;compares Register 0 with
        R 1      ;Register 1 and the the Status Flags
                 ;according to the result
```

Flags

ACCU		Unchanged		
		Value 1 = Value 2	Value 1 > Value 2	Value 1 < Value 2
Status Flags	P	High	High	Low
	Z	High	Low	Low
	N	Low	Low	High

See also
AND
OR
EXOR
FCMP

Practical example
Read two numbers; if the first number is greater, equal or lower than the second number then output 32, 33 or 34 respectively must be turned on.
The two numbers come from BCD encoders (2 digits) on inputs 16 to 23 and 24 to 31.

```
    COB     0
            0
    DIGI    2           ;read 2 digits
            I 16        ;from Input 16..32
            R 1         ;and store them in R 1
    DIGI    2           ;read 2 digits
            I 24        ;from input 24..31
            R 2         ;and store them in R 2
    CMP     R 1         ;compare R 1
            R 2         ;with R 2
    ACC     Z           ;if R 1 = R 2 (Z = 1)
    OUT     O 33        ;then set Output 33 and Flag 0
    OUT     F 0         ;else reset Output 33 and Flag 0
    ACC     N           ; if R 1 < R2
    OUT     O 34        ;then set output 34, else reset output 34
    ACC     P           ;if R 1 > R 2
    ANL     F 0         ;(use F 0 for "and not equal to")
    OUT     O 32        ;then set Output 32, else reset Output 32
    ECOB
```

5.3 DIV - Divide Register

Description
Signed integer division.
Divides the contents of the 1st Register or K constant by the contents of the 2nd Register or constant, and stores the result in the 3rd Register.
The remainder is placed in the 4th Register.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format
```
DIV[X]   [=] value1    (i)   ;R K
         [=] value2          ;R K
         [=] result    (i)   ;R
         [=] remainder (i)   ;R
```

Example
```
DIV      R 20       ;divide R 20
         K 1000     ;by 1000
         R 21       ;and put the result in Register 21
         R 22       ;and the remainder in Register 22
```

Flags

ACCU		Unchanged
Status Flags	E	Set on divide by zero

	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
FDIV

Practical example
Read two numbers, divide them and put the result in another Register.
The two numbers come from BCD encoders (2 digits) on Inputs 16 to 23 and 24 to 31.

```
    COB     0
            0
    DIGI    2        ;read 2 digits
            I 16     ;from Inputs 16..23
            R 1      ;and store them in R 1
    DIGI    2        ;read 2 digits
            I 24     ;from Inputs 24..31
            R 2      ;and store them in R 2
    DIV     R 1      ;R 100 = R 1 / R 2
            R 2
            R 100    ;result
            R 101    ;remainder
    CPB     E 99     ;if Error Flag is set,
                     ; then call program block 99
    ECOB

    PB 99
    SET O 47         ;alarm if division by zero (Output 47)
    EPB
```

5.4 MUL - Multiply Registers

Description
Signed integer multiplication.
Multiplies the contents of the 1st Register or K constant by the contents of the 2nd Register or K constant, and stores the result in the 3rd Register.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format
```
MUL[X]   [=] value1 (i)   ;R K
         [=] value2       ;R K
         [=] result (i)   ;R
```

Example
```
MUL     R 0    ;multiplies Register 0
        K 10   ;by 10
        R 0    ;and store the result in Register 0
```

Flags

ACCU		Unchanged
Status Flags	E	Set on overflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
FMUL

Practical example
Read two numbers, multiply them and put the result in another Register.
The two numbers come from BCD encoders (2 digits) on Inputs 16 to 23, and 24 to 31.

```
    COB     0
            0
    DIGI    2         ;read 2 digits
            I 16      ;from Inputs 16..23
            R 50      ;and store them in R 50
    DIGI    2         ;read 2 digits
            I 24      ;from Inputs 24..31
            R 55      ;and store them in R 55
    MUL     R 50      ;R 4000 = R 50 * R 55
            R 55
            R 4000    ;result
    ECOB
```

5.5 SQR - Square Root

Description
Integer Square Root.
The integer square root of the contents of the 1st Register is stored in the 2nd Register.
If the 1st Register contains a negative value, the Error flag is set and the operation is not performed.

Format
```
SQR[X]    [=] value  (i)   ;R
          [=] result (i)   ;R
```

Example
```
SQR     R 0     ;the square root of Register 0 is
        R 100   ;placed in Register 100
```

Flags

ACCU		Unchanged
Status Flags	E	Set on an attempt to obtain the square root of a negative value
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
FSQR

Practical example
Get the square root of a number read from BCD encoders (4 digits) on inputs 16 to 31.

```
    COB     0
            0
    DIGI    4        ;read 4 digits
            I 16     ;from Inputs 16..31
            R 100    ;and store them in R 100
    SQR     R 100    ;R 101 = square root of R 100
            R 101    ;result
    ECOB
```

5.6 SUB - Subtract Registers

Description
Signed integer subtraction.
Subtracts the contents of the 2nd Register or K constant from the contents of the 1st Register or K constant, and stores the result in the 3rd Register.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format
```
SUB[X]   [=] value1 (i)  ;R K
         [=] value2      ;R K
         [=] result (i)  ;R
```

Example
```
SUB     R 1    ;R 3 = R 1 - R 2
        R 2
        R 3
```

Flags

ACCU		Unchanged
Status Flags	E	Set on underflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
ADD
FSUB

Practical example
Read two numbers, subtract them and put the result in another register.
The two numbers come from BCD encoders (2 digits) on inputs 16 to 23, and 24 to 31.

```
    COB     0
            0
    DIGI    2       ;read 2 digits
            I 16    ;from Inputs 16..23
            R 10    ;and store them in R 10
    DIGI    2       ;read 2 digits
            I 24    ;from Inputs 24..31
            R 11    ;and store them in R 11
    SUB     R 10    ;R 12 = R 10 - R 11
            R 11
            R 12
    ECOB
```

5.7 UDIV - Unsigned Divide Register

Description
Unsigned integer division.
Divides the contents of the 1st Register or K constant by the contents of the 2nd Register or constant, and stores the result in the 3rd Register.
The remainder is placed in the 4th Register.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format
```
UDIV[X]  [=] value1    (i)   ;R K
         [=] value2          ;R K
         [=] result    (i)   ;R
         [=] remainder (i)   ;R
```

Example
```
UDIV     R 20      ;divide R 20
         K 1000    ;by 1000
         R 21      ;and put the result in Register 21
         R 22      ;and the remainder in Register 22
```

Flags

ACCU		Unchanged
Status Flags	E	Set on divide by zero
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
DIV
FDIV

5.8 UMUL - Unsigned Multiply Registers

Description
Unsigned integer multiplication.
Multiplies the contents of the 1st Register or K constant by the contents of the 2nd Register or K constant, and stores the result in the 3rd Register.
If a constant is used, it should have the 'K' data type. K constants can only be positive.

Format
```
UMUL[X]  [=] value1 (i)   ;R K
         [=] value2       ;R K
         [=] result (i)   ;R
```

Example
```
UMUL    R 0    ;multiplies Register 0
        K 10   ;by 10
        R 0    ;and store the result in Register 0
```

Flags

ACCU		Unchanged
Status Flags	E	Set on overflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
MUL
FMUL

6 Floating Point Instructions

Floating point values can only be stored in Registers or Data Blocks. They are loaded into Registers using the LD instruction.
To specify a floating point number, the number must include a decimal point '.' or an exponent 'E'. For example: 1.2, 1E3, 4.656E2.
By default, the PCD uses the "Motorola Fast Floating Point" (FFP) format for floating point numbers, but the latest PCD models also support IEEE Float and IEEE Double formats.

See Numeric Constants for details and ranges.

Important
Floating point values are stored in Registers in a special binary format, using this value as an integer will yield incorrect results.
Mixing integer and floating point values in arithmetic operations also gives invalid results.
The integer values must be converted to floating point and back with the IFP or FPI instructions, or one of the Macros described above.

Floating Point Formats
Each of the following instructions has a version for FFP (the default), IEEE Float and IEEE Double data.
For IEEE Float, precede the mnemonic with an 'E' character, for example `EIFP`, `EFADD` etc.
For IEEE Double, precede the mnemonic with 'D', for example: `DIFP`, `DFSUB`.

`IFP`	Integer to floating point
`FPI`	Floating point to integer
`FADD`	Floating point add
`FSUB`	Floating point subtract
`FMUL`	Floating point multiply
`FDIV`	Floating point divide
`FSQR`	Square root
`FCMP`	Floating point compare
`FSIN`	Sine function
`FCOS`	Cosine function
`FATAN`	Arc tangent
`FEXP`	Exponential function
`FLN`	Logarithm function
`FABS`	Absolute value

Special Operators and Macros for Floating Point handling
There are several "special operators" which can make IL programming easier.
These can be thought of a pre-defined Macros, and are resolved at assembly time, not at run time.

`@IEEE(value)`	Convert to IEEE Float
`@ISFLOAT(value)`	Returns 1 if value is floating point (Motorola FFP or IEEE)
`@IFP(int_value, expon`	Integer to FFP
`@FPI(ffp_value, expon`	FFP to integer
`@DFPHI(value)`	Returns upper DWORD of the IEEE double value of an IEEE Float or FFP float
`@DFPLO(value)`	Returns lower DWORD of IEEE double value of an IEEE Float or FFP float

`@IFPE(int_value, expo`	Returns IEEE float value for an integer: int * 10 ^ exponent
`@EFPI(ieee_value, exp`	Returns the integer value of an IEEE float

IEEE Float

To declare an IEEE Float, use the `I` postfix:

```
Symbol EQU 1.2I
...
LD  R 0
      1.23456I
```

See also @IEEE() - Convert to IEEE Float.

IEEE Double

Note: Double values cannot be assigned symbol names, because a symbol is a 32-bit value.

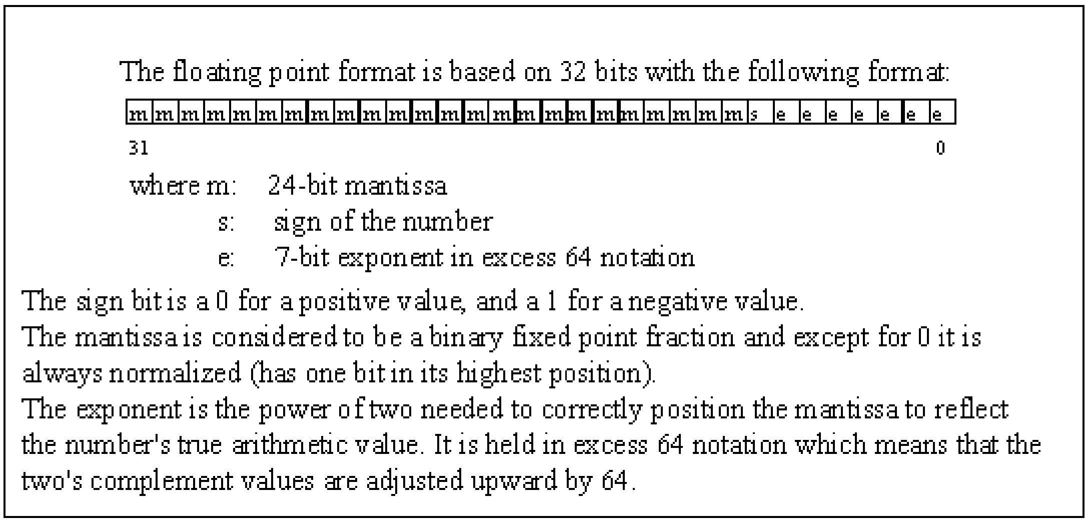
To declare an IEEE Double directly you can use an IL Macro like this:

```
;Load 2 registers with an IEEE Double value
DFLD MACRO reg, ffp_or_ieee_float
     LD R reg
        @DFPHI(ffp_or_ieee_float)
     LD R reg+1
        @DFPLO(ffp_or_ieee_float)
ENDM
...
DFLD(R 0, 1.2)    ;load R 0 and R 1 with Double value 1.2
```

@DFPHI() and @DFPLO() also accept IEEE or FFP symbols, their values are converted to double:

```
IEEESymbol EQU 1.2345678I  ;with 'I' postfix for IEEE float
FFPSymbol  EQU 1.2345678   ;the default is a Motorola Fast Floating Point (FFP) val
LD     R 100
       @DFPHI(IEEESymbol)   ;converts the IEEE value to double and returns the upper
LD     R 101
       @DFPLO(IEEESymbol)
LD     R 102
       @DFPHI(FFPSymbol)
LD     R 103
       @DFPLO(FFPSymbol)
```

Motorola Fast Floating Point (FFP) Format



The floating point format is based on 32 bits with the following format:

m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	s	e	e	e	e	e	e	e

31 0

where m: 24-bit mantissa
s: sign of the number
e: 7-bit exponent in excess 64 notation

The sign bit is a 0 for a positive value, and a 1 for a negative value.
The mantissa is considered to be a binary fixed point fraction and except for 0 it is always normalized (has one bit in its highest position).
The exponent is the power of two needed to correctly position the mantissa to reflect the number's true arithmetic value. It is held in excess 64 notation which means that the two's complement values are adjusted upward by 64.

6.1 DFPE - IEEE Double To Float

Description
Converts an IEEE double floating point value (64 bits) in two consecutive Registers into an IEEE float (single) value in one Register (32 bits).

Format
```
DFPE[X]  reg1 (i)   ;1st reg of pair with IEEE Double value to convert
         reg2 (i)   ;dest reg to receive the IEEE Float (single) value
```

Example
```
LD     R 100           ;load IEEE double into R 100 and R 101
       @DFPLO (1.234)  ;upper 32 bits in R 100
LD     R 101
       @DFPLO (1.234)  ;lower 32 bits in R 101
EFPD   R 100           ;convert IEEE Double in R 100 and R 101
       R 102           ;into IEEE Float 9single) in R 102
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the IEEE double is invalid or out of range
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
EFPD
@DFPHI() and @DFPLO()
IFP
FPI

6.2 EFPD - IEEE Float To Double

Description
Converts an IEEE floating point value (single) in a Register into an IEEE double value (64 bits) in two consecutive Registers.

Format
```
EFPD[X]  reg1 (i)   ;IEEE floating point value (single) to convert
         reg2 (i)   ;1st reg of pair to receive the IEEE Double value
```

Example
```
LD      R 100    ;R 100 = 1.234 IEEE float
        1.234I
EFPD    R 100    ;convert R 100 to IEEE Double
        R 101    ;in R 100 and R 101
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the IEEE floating point number is invalid
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
DFPE
IFP
FPI

6.3 FABS - Floating Point Absolute

Description
The absolute value (converted to positive if it is negative) of the 1st Register is stored in the 2nd Register.
The 1st Register must contain a valid floating point format value.

NT systems only: For IEEE Float use `EFABS`. For IEEE Double use `DFABS`.

Format
```
FABS[X]   [=] reg    (i)     ;R
          [=] result (i)     ;R
```

Example
```
FABS   R 1    ;R 2 = absolute value of R 1
       R 2    ;if R 1 contains -1.2 then R 2 = +1.2
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the floating point number is invalid
	P	Always set High
	Z	Set according to the result
	N	Always set Low

See also
"Advanced n-Dimensional Quantum Calculus For Busy Housewives", 42nd Edition

6.4 FADD - Floating Point Add

Description
Adds the contents of the 1st Register to the contents of the 2nd Register, and stores the result in the 3rd Register.
The Registers must contain valid floating point format values.

NT systems only: For IEEE Float use `EFADD`. For IEEE Double use `DFADD`.

Format

```
FADD[X]    [=] reg1   (i)    ;R
           [=] reg2          ;R
           [=] result (i)    ;
```

Example

```
FADD    R 100   ;R 500 = R 100 + R 101
        R 101
        R 500
```

Flags

ACCU		Unchanged
Status Flags	E	Set on overflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

ADD

6.5 FATAN - Floating Point Arc Tangent

Description

The arc tangent of the contents of the 1st Register is stored in the 2nd Register.
The 1st Register must contain a valid floating point value in RADIANS.
The result in the second Register will range from $-\Pi/2$ to $+\Pi/2$.

NT systems only: For IEEE Float use `EFATAN`. For IEEE Double use `DFATAN`.

Format

```
FATAN[X]   [=] reg    (i)   ;R
           [=] result (i)   ;R
```

Example

```
FATAN    R 1   ;R 0 = Arc tangent of value in R 1
         R 0
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the value in the first Register is too big
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

FSIN
FCOS

6.6 FCMP - Floating Point Compare

Description

Compares the contents of the 1st Register with the contents of the 2nd Register and sets the Status flags according to the result.
Neither of the Registers are altered.
Both Registers must contain valid floating point format values.

NT systems only: For IEEE Float use `EFCMP`. For IEEE Double use `DFCMP`.

Format

```
FCMP[X]    [=] reg1 (i)   ;R
           [=] reg2       ;R
```

Example

```
FCMP    R 0     ;compare R 0 and R 1
        R 1     ;set the status flags according to result
```

Flags

ACCU		Unchanged		
		Value 1 = Value 2	Value 1 >= Value 2	Value 1 < Value 2
Status Flags	E	Low	Low	Low
	P	High	High	Low
	Z	High	Low	Low
	N	Low	Low	High

See also

CMP

Note

Do not compare Floating Point values for equality, always use >= or < to avoid accuracy errors.

6.7 FCOS - Floating Point Cosine

Description

The cosine of the contents of the 1st Register is stored in the 2nd Register.
The 1st Register must contain a floating point value in RADIANS in the range of ±10^6

NT systems only: For IEEE Float use `EFCOS`. For IEEE Double use `DFCOS`.

Format

```
FCOS[X]   [=] reg   (i)     ;R
          [=] result(i)     ;R
```

Example

```
FCOS    R 100   ;R 20 = cosine of R 100
        R 20
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the value in the first Register is too big
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
FSIN
FATAN

6.8 FDIV - Floating Point Divide

Description
Divides the contents of the 1st Register by the contents of the 2nd Register, and stores the result in the 3rd Register.
Divide-by-zero sets the Error flag, and the operation is not performed.
Because Floating Point arithmetic is more exact than integer arithmetic, there is no remainder.

NT systems only: For IEEE Float use `EFDIV`. For IEEE Double use `DFDIV`.

Format
```
FDIV[X]     [=] reg    (i)   ;R
            [=] divisor      ;R
            [=] result (i)   ;R
```

Example
```
FDIV     R 1     ;R 3 = R 1 / R 2
         R 2
         R 3
```

Flags

ACCU		Unchanged
Status Flags	E	Set on divide by zero
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
DIV

6.9 FEXP - Floating Point Exponential

Description
Computes 'e' to the power of the contents of the 1st Register is stored in the 2nd Register.
The Register must contain a valid floating point format value.

NT systems only: For IEEE Float use `EFEXP`. For IEEE Double use `DFEXP`.

Format
```
FEXP[X]      [=] reg    (i)   ;R
             [=] result (i)   ;R
```

Example
```
FEXP    R 0     ;R 1 = e ^ R 0
        R 1
```

Flags

ACCU		Unchanged
Status Flags	E	Set on overflow
	P	Set according to the result
	Z	Set according to the result

N   Set according to the result

See also
FPI
IFP

6.10 FLN - Floating Point Logarithm

Description
The natural log of the contents of the 1st Register is stored in the 2nd Register.
The 1st Register must contain a valid floating point format value.
If the natural log of a negative value is taken, the Error flag is set and the log of the absolute (+ve) value is taken.

NT systems only: For IEEE Float use `EFLN`. For IEEE Double use `DFLN`.

Format
```
FLN[X]     [=] reg    (i)    ;R
           [=] result (i)    ;R
```

Example
```
FLN    R 1    ;R 2 = ln R 1
       R 2
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the "ln" of zero or a negative value is taken
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
FEXP

6.11 FMUL - Floating Point Multiply

Description
Multiplies the contents of the 1st Register by the contents of the 2nd Register, and stores the result in the 3rd Register.
Both Registers must contain valid floating point format values.

NT systems only: For IEEE Float use `EFMUL`. For IEEE Double use `DFMUL`.

Format
```
FMUL[X]   [=] reg1   (i)   ;R
          [=] reg2         ;R
          [=] result (i)   ;R
```

Example
```
FMUL      R 20    ;R 0 = R 20 * R 30
          R 30
          R 0
```

Flags

ACCU	Unchanged

Status Flags	E	Set on overflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
MUL

6.12 FPI - Floating Point to Integer

Description
Converts the floating point value in the specified Register to integer format.
The 2nd operand indicates the power of ten to be used in the conversion.
The result is the integer of the result of the Register contents multiplied by 10 to the power of the 2nd operand.

For example, if the Register contains 1234.56 and the power of ten is 2, the integer result will be 12.
If the conversion is not possible, the Error flag is set and nothing is done.

NT systems only: For IEEE Float use `EFPI`. For IEEE Double use `DFPI`.

Format
```
FPI[X]     [=] reg (i)    ;R
           power          ;power of ten -20 to +18
```

Example
```
FPI     R 0  ;if R 0 contains 1234.56, it is converted
        0    ;to the integer value 1234 (power of ten is zero)
```

Flags

ACCU		Unchanged
Status Flags	E	Set on overflow
	P	Unchanged
	Z	Unchanged
	N	Unchanged

See also
IFP
@FPI( )

Practical example

R 500 Before	Instruction		Conversion	R 500 After

`123.456`	`FPI`	`R 500`
		`0`
			`R 500 * 10^0`	`123`

`123.456`	`FPI`	`R 500`
		`-2`
			`R 500 * 10^-2`	`1`

`123.456`	`FPI`	`R 500`
		`3`
			`R 500 * 10^3`	`123456`

6.13 FSIN - Floating Point Sine

Description

The sine of the contents of the 1st Register is stored in the 2nd Register.
The 1st Register must contain a floating point value in RADIANS in the range ±10^6.

NT systems only: For IEEE Float use `EFSIN`. For IEEE Double use `DFSIN`.

Format

```
FSIN[X]    [=] reg    (i)    ;R
           [=] result (i)    ;R
```

Example

```
FSIN     R 0     ;R 100 = Sine of R 0
         R 100
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the value in the first Register is too big
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also

FCOS
FATAN

6.14 FSQR - Floating Point Square Root

Description

Stores the square root of the contents of the 1st Register into the 2nd Register.
If the 1st Register contains a negative value, the Error flag is set and the square root of the absolute (+ve) value is taken.

NT systems only: For IEEE Float use `EFSQR`. For IEEE Double use `DFSQR`.

Format

```
FSQR[X]    [=] reg    (i)   ;R
           [=] result (i)   ;R
```

Example

```
FSQR     R 0   ;R 0 = Square root of R 0
         R 0
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the value was negative
	P	Always set High
	Z	Set according to the result
	N	Always set Low

See also

SQR

6.15 FSUB - Floating Point Subtract

Description
Subtracts the contents of the 2nd Register from the contents of the 1st Register, and stores the result in the 3rd Register.
Both Registers must contain valid floating point format values

NT systems only: For IEEE Float use `EFSUB`. For IEEE Double use `DFSUB`.

Format
```
FSUB[X]   [=] reg1   (i)    ;R
          [=] reg2          ;R
          [=] result (i)    ;R
```

Example
```
FSUB    R 0   ;R 0 = R 0 - R 1
        R 1
        R 0
```

Flags

ACCU		Unchanged
Status Flags	E	Set on underflow
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
SUB

6.16 IFP - Integer to Floating Point

Description
Converts the integer value in the specified Register to floating point format.
The 2nd operand indicates the power of ten to which the integer is to be raised, this controls the position of the decimal point.
For example, if the power of ten is +3, the contents of the Register is multiplied by 1000 (10^3), and the result is stored in the Register in floating point format.
If the Register contained 12, the result would be 12000.00.
If the conversion is not possible (number too big or too small), the Error flag is set and no conversion is done.

NT systems only: For IEEE Float use `EIFP`. For IEEE Double use `DIFP`.

Format
```
IFP[X]   [=] reg (i)    ;R
         power          ;power of ten -20 to +18
```

Example
```
IFP     R 0    ;R 0=floating point value of
        3      ;R 0 * 10^3
```

Flags

ACCU		Unchanged
Status Flags	E	Set if conversion is not possible
	P	Unchanged
	Z	Unchanged

N Unchanged

See also

FPI

@IFP( )

Practical example

R 500 Before	Instruction	Conversion	R 500 After
`123`	`IFP R 500 0`	`R 500 * 10^0`	`1.23E+2`
`123`	`IFP R 500 -2`	`R 500 * 10^-2`	`1.23E+0`
`123`	`IFP R 500 3`	`R 500 * 10^3`	`1.23E+5`

7 Bloctec Instructions

Bloctec is a structured programming method which breaks a program down into separate blocks of code.
A Cyclic Organization Block (COB) is the main task, which will typically call Program Blocks (PB) and Function Blocks (FB), or Graftec Sequential Blocks (SBs), up to a call-nesting depth of 7.

At least one COB, usually COB 0, must be present in the program. Only Function Blocks (FBs) can have run-time parameters.

For more information about the structured programming methods refer to "Structured Programming" in the User's Guide.

`COB`	Cyclic Organization Block
`ECOB`	End Cyclic Organization Block
`XOB`	Exception Organization Block
`EXOB`	End Exception Organization Block
`PB`	Program Block
`EPB`	End Program Block
`CPB`	Call Program Block
`CPBI`	Call Program Block Indirect
`FB`	Function Block, with optional parameters
`EFB`	End Function Block
`CFB`	Call Function Block
`NCOB`	Next Cyclic Organization Block
`SCOB`	Stop Cyclic Organization Block
`CCOB`	Continue Cyclic Organization Block
`RCOB`	Restart Cyclic Organization Block

Important
Calling the same FB or PB from different COBs or different places in the program may have unexpected results if the data it uses is also shared - if the same global data are used.
To allow code sharing, or different "instances" of a block which is like a re-usable component, the data must be supplied via FB parameters or indirectly via a base Register or Data Block.

The following instructions must NEVER be used in a Graftec program because they can compromise event synchronization:
RCOB, NCOB, SCOB, CCOB and the COB instruction's supervision time.

7.1 CCOB - Continue Cyclic Organization Block

Description
Conditionally or unconditionally allows a COB that was stopped by the SCOB instruction to resume execution.
If the condition is not satisfied, the COB is not resumed.
CCOB does not cause the COB to be executed immediately, but allows it to be executed the next time it is scheduled.

Tip: Well-structured programs should not need this instruction. It should only be used in your application with the utmost care.

This instruction must not be used in a Graftec program because it can destroy event synchronization.

Condition	Executed
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format

```
CCOB    [cc]  cob    ;COB
                     ;cc = condition code: H | L | P | N | Z |E
```

Example

```
CCOB    L 10   ;COB 10 is resumed if the ACCU is Low (0)
CCOB    0      ;COB 0 is resumed unconditionally
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

NCOB
RCOB
SCOB

7.2 CFB - Call Function Block

Description

Conditionally or unconditionally calls a Function Block. If the condition is not satisfied, the FB is not called.
An optional parameter list can follow the CFB instruction. The parameters are used by instructions within the Function Block.
Parameters are referenced by using '= n' as the operand, where 'n' is the parameter number to use (1-255).
The value of this parameter is substituted as the operand.

Condition	Executed
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

NOTE

CFB parameters are 16-bit values. The 32-bit LD instruction value cannot be supplied as a Function Block parameter. But you can use LDH and LDL to load a 16-bit Function Block parameter into the upper or lower 16 bits of a Register, Timer or Counter. Or pass the 32-bit value in a Register.

Format

```
CFB    [cc] number    ;FB, cc = condition code: H|L|P|N|Z|E
       [param 1]      ;optional parameter list
```

```
        [param 2]
        ...
        [param n]
        ...
```

Example

```
CFB    H 10     ;calls FB 10 if the ACCU is High
      32        ;parameter 1
      R 10      ;parameter 2
```

Flags

ACCU	Set High (1) at the start of the CFB When the program returns from the FB, the ACCU is restored to the state it had before the FB was called.
Status Flags	Depend on the FB code, they are not restored to the state they had before the FB was called.

See also

FB
CPB
PG5 User's Guide

7.3 COB - Cyclic Organization Block

Description

Starts the specified Cyclic Organization Block. The 2nd operand is the COB supervision time, in 10 millisecond increments.
If the supervision time elapses before the COB has finished execution (ECOB reached), the Exception XOB 11 is executed if it is present; if not present, the next COB is started.
If the supervision time is 0, XOB 11 is never executed, the next COB is started only when this COB has ended (the ECOB is reached).
If several COBs are programmed, they run one after the other in numerical order.

The ACCU is always set High (1) at the start of each COB.

The COB instruction uses 3 program lines because the supervision time needs 32 bits.

Format

```
COB     number     ;COB
        time       ;supervision time in 10ms increments
```

Example

```
COB     0          ;start of COB 0
        0          ;supervision time = 0
...
;body of COB 0
ECOB               ;end of COB 0
```

Flags

ACCU	Set High at start of COB.
Status Flags	Unchanged

See also

ECOB
NCOB
RCOB

SCOB
XOB

7.4 CPB - Call Program Block

Description
Conditionally or unconditionally calls a Program Block. If the condition is not satisfied, the PB is not called.

Condition	Executed
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format
```
CPB    [cc] number ;PB number, cc = condition code: H | L | P | N | Z | E
```

Example
```
CPB    10    ;unconditionally call PB 10
```

Flags

ACCU	Set High for the start of the PB. When the PB returns, the ACCU is restored to the state it had before the call.
Status Flags	Depend on the PB code, they are not restored to the state they had before the PB was called.

See also
PB
EPB
CFB

Practical example
IF.. THEN.. ELSE structure:

```
    COB     0
            0
    ...
    STH     I 15        ;if Input 15 is High
    CPB     H 20        ;then call PB 20
    CPB     L 25        ;else call PB 25
    ...
    ECOB

    PB 20
    ...
    EPB

    PB 25
    ...
    EPB
```

7.5 CPBI - Call Program Block Indirect

Description

Conditionally or unconditionally calls a Program Block whose number is contained in the given Register.
Since this instruction uses a condition code, the 'R' data type is not required.
If the given Register contains an invalid PB number, or the PB does not exist, the Error flag is set and XOB 13 is called (if present).
If the condition is not satisfied, the PB is not called.

Notes

- For firmware versions earlier than 1.20.00, the max. Register address for indirect instructions is 8191.
- To use Register addresses 8192..16383 with firmware version 1.2.00 or later, set the Build Option "Use 16-bit Register and Flag addressing" to Yes.
- This instruction cannot be used with Function Block parameters (= n).
- Temporary Registers, defined with TEQU, cannot be used.

Condition	**Executed**
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format

```
CPBI     [cc]  reg     ;reg = Register number containing
                       ;the number of the PB to be called
                       ;cc = condition code: H | L | P | N | Z | E
```

Example

```
CPBI     L 10     ;if ACCU is Low (0), then the PB whose
                  ;number is in R 10 is called
```

Flags

ACCU	Set High for the start of the PB. When the PB returns, the ACCU is restored to the state it had before the call.
Status Flags	Depend on the PB code, they are not restored to the state they had before the PB was called.

See also

PB
EPB
CFB

7.6 ECOB - End Organization Block

Description

Ends the current COB. The next COB (if present) will begin execution.
A COB body must always be terminated by an ECOB instruction.

Format

```
ECOB     ;no operands
```

Example
```
COB      0    ;start of COB 0
         0    ;supervision time
...           ;body of COB
ECOB          ;end of COB
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
COB

7.7 EFB - End Function Block

Description
Ends the current Function Block (FB).
Returns to the instruction following the Call Function Block (CFB) instruction.

Format
```
EFB    ;no operands
```

Example
```
FB     3   ;start of FB 3
...        ;body of FB 3
EFB        ;end of FB 3
```

Flags

ACCU	Restored to the state it had before the FB was called.
Status Flags	Unchanged

See also
FB
CFB

7.8 EPB - End Program Block

Description
Ends the current Program Block (PB).
A return is made to the instruction after the Call Program Block (CPB) instruction.

Format
```
EPB    ;no operands
```

Example
```
PB     6    ;start of PB 6
...         ;body of PB 6
EPB         ;end of PB6
```

Flags

ACCU	Restored to the state it had before the PB was called.
Status Flags	Unchanged

See also
PB

CPB

7.9 EXOB - End Exception Organization Block

Description
Ends the current XOB. At the EXOB instruction, the XOB returns to the location from where it was called.

Format
```
EXOB     ;no operands
```

Example
```
XOB     16    ;start of XOB 16
...           ;body of XOB 16
EXOB          ;end of XOB 16
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
XOB
XOB List

7.10 FB - Function Block

Description
Begins a Function Block (FB). An FB is a subroutine with optional parameters.
A list of FB parameters can be defined, this list is supplied when the FB is called.

Format
```
FB    number
```

Example
```
FB    10     ;start of FB 10
...
STH   =1     ;FB parameter reference, parameter 1
...
EFB          ;end of FB 10
```

Flags

ACCU	Set High at start of FB, restored at end of FB
Status Flags	Unchanged

See also
EFB
CFB

Practical example
Compute the formula: Z = X * (X+Y)

```
    FB     25       ;Function Block  X * (X+Y)
    ADD    =1       ;Z = X + Y
           =2
           =3
    MUL    =3       ;Z = Z * X
           =1
           =3
    EFB

    COB    7
           0
    ...
    STH    I 1      ;if Input 1 goes High
    DYN    F 1
    CFB    H 25     ;then R107 = R100 * (R100+330)
           R 100    ;parameter 1  (X)
           K 330    ;parameter 2  (Y)
           R 107    ;parameter 3  (Z)

    STH    I 2      ;if Input 2 goes H
    DYN    F 2
    CFB    H 25     ;then R107 = R200 * (R200+R201)
           R 200    ;parameter 1  (X)
           R 201    ;parameter 2  (Y)
           R 107    ;parameter 3  (Z)
    ...
    ECOB
```

7.11 NCOB - Next Cyclic Organization Block

Description
Conditionally or unconditionally forces the program to switch to the next COB.
If the condition code is not satisfied, the NCOB instruction is ignored.
Wait loops can be programmed using NCOB without interfering with the execution of any other COBs.
For every wait loop, an NCOB instruction should be inserted. This allows "parallel" execution of COBs.

Tip: Good Bloctec programs should not include wait loops, and therefore should not need to use NCOB.
Programs should normally use the ACCU state to control program execution.

This instruction must not be used in a Graftec program because it can destroy event synchronization.

Condition	Executed
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format
```
NCOB   [cc]   ;cc = condition code H | L | P | N | Z | E
```

Example
```
STH    I 15    ;wait until I 15 = L
NCOB   L       ;ensure all other tasks execute
JR     L -2
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
RCOB
SCOB
CCOB

7.12 PB - Program Block

Description
Marks the beginning of a Program Block (PB). A Program Block is a subroutine without parameters.

Format
```
PB      number
```

Example
```
PB    26  ;start of PB 26
...       ;body of PB 26
EPB       ;end of PB 26
```

Flags

ACCU	Set High at start of PB, restored at end of PB
Status Flags	Unchanged

See also
EPB
CPB
FB

7.13 RCOB - Restart Cyclic Organization Block

Description
Restarts any COB, conditionally or unconditionally, from the given program line.
This instruction can be used within any COB or XOB. If the condition is not satisfied, the RCOB instruction is ignored.
The 1st operand is the COB number to be restarted.
The 2nd operand is the program line number to restart from.
The line number is an offset from the start of the COB, it is NOT an absolute program line number.

This instruction must not be used in a Graftec program because it can destroy event synchronization.

Condition	Executed
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format
```
RCOB     [cc] cob      ;condition code + COB number
```

```
         line          ;line number from start of COB
                       ;cc = condition code: H | L | P | N | Z | E
```

Example

```
RCOB     0       ;restarts COB 0
         10      ;execution begins from line 10 of COB 0
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

NCOB
SCOB
CCOB

7.14 SCOB - Stop Cyclic Organization Block

Description

Stops the given COB conditionally or unconditionally. Execution continues with the next COB.
The COB is not executed again until the correct CCOB instruction is executed by another COB.
A COB can stop itself executing, but must be restarted by another COB containing a CCOB instruction.
If the condition is not satisfied, the SCOB instruction is ignored.

Tip: Well-structured programs should not need this instruction. It should only be used in your application with the utmost care.
Good Bloctec programs should not use any wait loops, and therefore should not need to use NCOB.
Programs should normally use the ACCU status to control program execution.

This instruction must not be used in a Graftec program because it can destroy event synchronization.

Condition	**Executed**
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format

```
SCOB   [cc] cob    ;condition code + COB number
                   ;cc = condition code: H | L | P | N | Z | E
```

Example

```
SCOB   L 10    ;stops COB 10 if ACCU is Low
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

CCOB
NCOB
RCOB

7.15 XOB - Exception Organization Block

Description
Marks the beginning of an Exception Organization Block (XOB).
An XOB is called when an error or another important event occurs. The XOB can contain program code to handle these events.
If no associated XOB is present, no action is taken (the event is ignored) and the Error lamp may be turned on if it was an error event.
At the end of the XOB, the exception routine will return to the location from where it was called.

Format
```
XOB   number
```

Example
```
XOB    16        ;startup XOB
...              ;body of XOB
EXOB             ;end of XOB
```

Flags

ACCU	Set High at start of XOB, restored at end of XOB
Status Flags	Unchanged

See also
EXOB
XOB List

8 Graftec Instructions

Saia PG5 Graftec is a graphical programming method for sequential (step-by-step) processes.

A Graftec program consists of a sequence of alternating Steps (ST) and Transitions (TR), encapsulated in a Sequential Block (SB).
The SB is called repeatedly from a Cyclic Organization Block (COB), and it executes only the pending Transitions (checks for pending events) and executes the associated STep if the Transition was satisfied.

Steps contain actions to be performed, instructions such as SET, RES, STXT, etc. Transitions contain conditional linkages using instructions such as STH, ANL CMP, etc.
A TR must always be followed by an ST. The ST is executed only if the preceding TR is satisfied (ACCU is High at ETR).

Graftec programs are usually written using the Saia PG5 Graftec Editor (S-Graf). This editor automatically handles the program structure, which you create graphically on the screen.
With this editor you don't need to use the low-level Graftec instructions listed below.

For more information about Graftec programming refer to the "Structured Programming" chapter in the PG5 User's Guide.

SB	Sequential Block
ESB	End Sequential Block
IST	Initial Step
ST	Step
EST	End Step
TR	Transition
ETR	End Transition
CSB	Call Sequential Block
RSB	Restart Sequential Block

8.1 CSB - Call Sequential Block

Description
Conditionally or unconditionally calls a Sequential Block. If the condition is not satisfied, the SB is not called.
A sequential block cannot be called from another SB.

Condition	**Executed**
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Format
```
CSB     [cc] number      ;SB number
                         ;cc = condition code: H | L | P | N | Z | E
```

Example

```
CSB     L 10    ;call SB 10 if the ACCU is Low
```

Flags

ACCU	Set High for the start of the SB. When the SB returns, the ACCU is restored to the state it had before the call.
Status Flags	Depend on the SB code, they are not restored to the state they had before the SB was called.

See also

SB

CPB

8.2 ESB - End Sequential Block

Description

Ends the current Sequential Block (SB).

Format

```
ESB   ;no operand
```

Example

```
SB     10     ;start of SB 10
...           ;body of SB, contains STs and TRs
ESB           ;end of SB 10
```

Flags

ACCU	Restored to the state it had before the SB was called.
Status Flags	Unchanged

See also

SB

ST

TR

8.3 EST - End Step

Description

Ends the current Step or Initial Step (ST or IST)

Format

```
EST        ;no operand
```

Example

```
ST     0      ;start of ST 0
       I 25   ;incoming from Transition 25
       O 47   ;outgoing to Transition 47
...           ;body of ST 0
EST           ;end of ST 0
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

ST

TR
SB

8.4 ETR - End Transition

Description
Ends the current Transition (TR).

Format
```
ETR     ;no operand
```

Example
```
TR      0       ;start of TR 0
        I 12    ;incoming from Step 12
        O 14    ;outgoing to Step 14
...             ;body of TR 0
ETR             ;end of TR 0
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
TR
ST
SB

8.5 IST - Initial Step

Description
The Initial Step defines the first Step to be executed when a Sequential Block (SB) is called.
Every SB must have at least one Initial Step. In all other respects the Initial Step is the same as any other Step (see ST).
IST is followed by a list of incoming (I) and outgoing (O) Transitions.

Format
```
IST     number     ;initial Step number
        list       ;incoming and outgoing transitions list
                   ;(variable length)
```

Example
```
IST     1        ;Initial Step 1
        I 900    ;incoming from Transition 900
        O 1      ;outgoing to Transition 1
...              ;body of ST 1
EST              ;end of ST 1
```

Flags

ACCU	Set High at the start of the Initial Step
Status Flags	Unchanged

See also
EST
SB
ST

8.6 RSB - Restart Sequential Block

Description
Conditionally or unconditionally restarts a Sequential Block (SB).
The 1st operand is the number of the SB to be restarted.
The 2nd operand is the STep number from where the SB is to be restarted, or a list of Steps if in a parallel branch.
The Steps must be in the SB to be restarted!

If the restart must take place in simultaneous branches (parallel programs), the "RSB" instruction will contain as many additional lines as steps to be restarted.

Condition	Executed
blank	Always (no condition code)
H	If Accumulator = H (1)
L	If Accumulator = L (0)
P	If Positive flag = H (Negative flag = L)
N	If Negative flag = H
Z	If Zero flag = H
E	If Error flag = H

Tips: The RSB instruction can be used to restart a Graftec program from a pre-defined point. This could be useful if a long process must be interrupted for some reason, such as a power down. However, the Steps from where the Graftec will restart must be programmed in advance, the PCD does not save the state of the Graftec.
This means that the process can only be stopped and restarted at certain pre-programmed points, and the code to handle this must be hand-written in IL.
You must also be sure that all data is properly initialized or is non-volatile.
Note that Timers and other data which is set to zero by a restart will need special attention.
One way to approach this would be to store a value in a Register which indicates the current state, and use this to select the RSB instruction with the correct list of Steps from where the process will be restarted.

The RSB instruction is potentially very dangerous. The assembler cannot verify the parameters are correct. If incorrect it can cause fatal de-synchronization of the Graftec program.

Format
```
RSB     [cc] number    ;SB number
                       ;cc = condition code: H | L | P | N | Z | E
        step           ;STep number
        [step]         ;[STep number]
        ...
```

Example
```
RSB     12      ;restarts SB 12
        11      ; at Step 11
```

Flags

ACCU	Set High before restarting the SB
Status Flags	Unchanged

See also
SB
CSB
ST

8.7 SB - Sequential Block

Description

Starts a Sequential Block (SB). A Sequential Block contains one independent Graftec program. The SB contains the Steps and Transitions, and their code in IL or Fupla.

Format

```
SB  number
```

Example

```
SB      10      ;start of SB 10
...             ;body of SB 10, contains STs and TRs
ESB             ;extra special beer
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

ESB
CSB
RSB
IST
ST
TR

8.8 ST - Step

Description

Defines the start of a Step (ST). Following the ST instruction must be a list of incoming (I) and outgoing (O) Transitions.
A Step should typically contain only action instructions such as SET, RES, OUT, LD, MOV, FADD, etc. It can also call FBs and PBs.
In Saia PG5 Graftec, once a Step has been executed, the program pointer moves to the next Transition.
Steps can only appear inside SBs.

Steps should not contain wait loops are call blocks which contain wait loops.

Format

```
ST     number       ;Step number
       list         ;incoming and outgoing Transition list
                    ;(variable length)
```

Example

```
ST     10     ;Step 10
       I 9    ;incoming from Transition 9
       O 10   ;outgoing to Transition 10
...           ;body of Step
EST           ;end of Step
```

Flags

ACCU	Set High at the start of the ST
Status Flags	Unchanged

See also
EST
IST
TR
SB

8.9 TR - Transition

Description
Defines the start of a Transition (TR). Following the TR instruction must be a list of all the incoming (I) and outgoing (O) Steps.
Typically a Transition should contain logical instructions forming a linkage whose final result indicates whether the following Step is to be executed.
If the final result of the Transition is false (ACCU = L (0)), then the next Step is not executed, execution continues with the next parallel branch or COB.
The next time the TR executes, the whole Transition is processed again.
The next Step is executed only if the final result of the Transition is TRUE (ACCU = H (1)).
With OR branching, the order of handling of the parallel TRs is set by the order of the outgoing Transitions defined in the preceding Step.
TRs can only appear inside SBs.

TRs should not contain wait loops, or call blocks which contain wait loops.

Format
```
TR    number    ;Transition number
      list      ;incoming and outgoing Steps list
                ;(variable length)
```

Example
```
TR    10        ;Transition number 10
      I 900     ;incoming from Step 900
      O 1       ;outgoing to Step 1
      O 2       ;outgoing to Step 2
      ...       ;body of TR 10
ETR             ;end of Transition 10
```

Flags

ACCU	Set High at the start of the ST
Status Flags	Unchanged

See also
ETR
SB
ST

9 Communications Instructions

Networks
Automation solutions often consist of several decentralized PCD controllers, terminals and supervision computers, connected by a communications network. Each station controls part of the process, and exchanges data with the other stations on the network. To guarantee the flexibility of such a concept, the PCD system supports several types of communications network. Each network has its own capabilities, so the user should choose the network which is most appropriate for the application.

The Saia PG5® is an effective tool for implementing these solutions:

- Saia PG5 Project Manager provides an overview of the stations (PCDs) and their configuration parameters including the network's communications parameters.
- The Fupla and IL editors allow the programming of the data exchange between PCD stations on the network.

The choice of network depends on the application's requirements.

These are the available network types:

Profi-S-Bus	Fieldbus network based at the Profibus FDL standard
Ether-S-Bus	Information network based on the standard Ethernet
Serial S-Bus	Network based on serial interface RS 485/232
S-Bus-Modem	Network based on analogue or digital telephone line
Profibus DP	Fieldbus network based on the standard Profibus DP
Profi-S-IO	Fieldbus network based on the standard Profibus DP

The different networks are distinguished by their services, technical characteristics and their application domains.

Although all the communication networks support the transport of PCD data as Inputs, Outputs, Flags, Registers etc., some also support the programming, control and commissioning of the PCD systems through the network using the PG5 tools.

Open protocol
Serial communications Mode C allows the exchange characters and strings without any specific protocol.
This mode is often used to support a text terminals or to implement another communication protocol with a non-Saia PCD device.

Old protocols
The old Mode D and MM4 are not supported by the new PCD models, and should not be used.

9.1 Mode C

Sends or receives single characters, or transmits a Text.

- Single characters from a Register or a Text are output.
- Single characters can be received and transferred into a Register.
- Often used to communicate with a terminal or printer.
- Can be used to implement custom messages and protocols.

See:

SASI Text (Mode C)
SRXD Receive Character (Mode C)
STXD Transmit Character (Mode C)
STXT Transmit Text (Mode C)
Texts Containing Data (Mode C)
Text Output Formats (Mode C)

9.2 Mode D

Old protocol, not supported by new PCD models.

Data Mode. Uses telegrams in accordance with ISO 1745, IBM BSC and DIN 66019.
Data can be exchanged between two PCDs or between a PCD and another intelligent system (IBM PC, etc) connected directly or via the Saia LAN 1.
The data can be the state of Inputs, Outputs or Flags; or the contents of Registers, Timers or Counters.

<mode>	Description
`MD0`	Mode D master
`SD0`	Mode D slave

The two modes are almost the same; the only difference is that when a conflict occurs in the full duplex communication, the Master station always has priority over the Slave to repeat the request. When communicating with a PC, the PCD must be set as Slave (SD0).

For a description of the complete protocol, consult the "Functional specification for the Saia P8 Protocol".

See:
SASI Text (Mode D & MM4)
SRXM Receive Media (Mode D)
STXM Transmit Media (Mode D)

9.3 Mode MM4

Old protocol, not supported by new PCD models.

Mode MM4 allows the connection of the PCD on the COMPEX LAC/LAC2 Network.
The LAC/LAC2 is an industrial local area network which supports the connection of different intelligent devices.
The PCD is connected to the network via a communicator which provides the required transmission services.
MM4 mode exchanges 32-bit Registers and up to 64 Registers can be transferred with one telegram.
This mode also supports a point-to-point connection between two PCDs.

See:
SASI Text (Mode D & MM4)
SRXM Receive Media (Mode MM4)
STXM Transmit Media (Mode MM4)

9.4 Serial-S-Bus

The Serial-S-Bus Master/Slave data exchange between PCDs or PLCs and PCDs.
The PG5 programming tool supports the full maintenance of the all PCDs present on a network.
The connection can be a point-to-point RS-232, a network RS-485, or remote communication with a modem.
Serial S-Bus supports only one master per network, but multi-master can be provided by using the S-Bus Gateway.
Communications is done using the STXM and SRXM instructions.

Characteristics

Max. transmission speed	115 KB
Exchange mode	Master/slave
Max. number of stations	255

Instructions
SASI Assign Serial Interface
SASII Assign Serial Interface Indirect
SASI Text (Serial S-Bus)
SRXM Receive Media (Mode S-Bus)
SRXMI Receive Media indirect (Mode S-Bus)
STXM Transmit Media (Mode S-Bus)
STXMI Transmit Media Indirect (Mode S-Bus)
SICL Serial Input Control Line
SOCL Serial Output Control Line

See also
For more information, consult the S-Bus Manual 26/739.

9.5 Profi-S-Bus

Profi-S-Bus supports multi-master data exchange between PCDs or PCDs and other PLC's connected on the network.
The PG5 programming tools support the commissioning and maintenance of the all PCDs on the network.

The master-master functionality is an improvement on the standard Serial-S-Bus protocol.
Standard Serial-S-Bus only allows one master per network. In a Profi-S-Bus network, all stations can be masters.

Communication is done using the familiar STXM and SRXM instructions. The syntax is similar to that of existing Serial S-Bus, but the station numbering is different.

Characteristics

Max. transmission speed	12 Mbd .
Exchange mode	Multi-Master
Max. number of stations	126

Instructions
SASI Assign Serial Interface
SASII Assign Serial Interface Indirect
SASI Text (Profi-S-Bus)
SRXM Receive Media (Mode S-Bus)

SRXMI Receive Media indirect (Mode S-Bus)
STXM Transmit Media (Mode S-Bus)
STXMI Transmit Media Indirect (Mode S-Bus)

9.6 Ether-S-Bus

Ether-S-Bus supports a multi-master data exchange between PCDs or PCDs and other PLCs connected on the network.
The PG5 programming tools support the commissioning and maintenance of the all PCDs present on the Ethernet network.

The master-master functionality in one network is an improvement on the standard Serial S-Bus protocol.
Standard Serial-S-Bus only allows one master per network. In an Ether-S-Bus network all stations can be masters.

Communication is done using the familiar STXM and SRXM instructions. The syntax is similar to that of existing Serial S-Bus, but the station numbering is different.

Characteristics

Communication security	Maximum 3 retries in background. S-Bus CRC 16 error checking is applied. No special secure layer via IP is used
Protocol via IP	The UDP protocol is used for communication in S-Bus via Ethernet. The communications socket is open and permanently tied to port 5050.
Max. transmission speed	10 and 100 Mbd
Exchange mode	Multi-Master
Max. number of stations	Unlimited: not limited

Instructions
SASI Assign Serial Interface
SASII Assign Serial Interface Indirect
SASI Text (Ether S-Bus)
SRXM Receive Media (Mode S-Bus)
SRXMI Receive Media indirect (Mode S-Bus)
STXM Transmit Media (Mode S-Bus)
STXMI Transmit Media Indirect (Mode S-Bus)

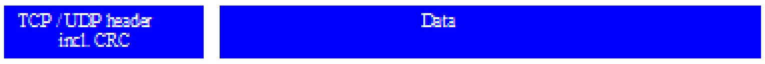
Programming open data mode via Ethernet
Open data mode is used for PCD communication with a foreign device that does not support S-Bus. However, it is equally possible for two PCDs to communicate together in open data mode, if required. Foreign devices do not support proprietary protocols (i.e. S-Bus). Therefore only raw data blocks (chars, strings, without header) should be transferred via IP.

The PCD can send data to a remote station, but in client mode it cannot directly request data from the remote station. Data received in open data mode is transmitted to the application layer.

If the transport protocol is UDP, it is not possible to recognize whether the connection between two stations has been broken. This feature must be implemented by the user in the application layer.
A communication control mechanism that recognizes communication breaks has been implemented within the TCP protocol.

Description of open data mode

In open data mode, raw data is attached to the UDP or TCP header and then transmitted. In S-Bus via UDP, data is always attached to the UDP header. In open data mode UDP, the maximum admissible length of transmitted data is 2048 bytes per datagram.



Configuration

The IP module must be configured with the IP address, subnet mask and default router as S-Bus via IP using the PG5's Device Configurator. No further configuration is necessary. Open data mode is initialized with the InitODM command.

Programming with IL

Open data mode is programmed by calling System Functions. System Functions are like FB calls, and are supplied as libraries. There is a library for the Ethernet TCP/IP module. This library is included in the FW. Functions are called with the CSF instruction.

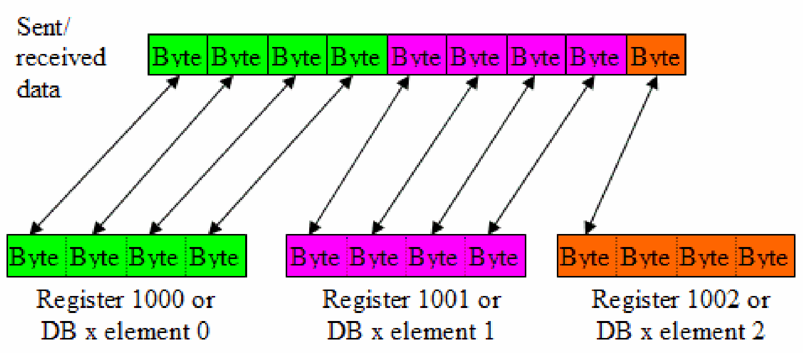
See the "SF IP Library" help, which can be displayed from S-Edit's "Function Selector" window, or from Library Manager.

Byte swapping

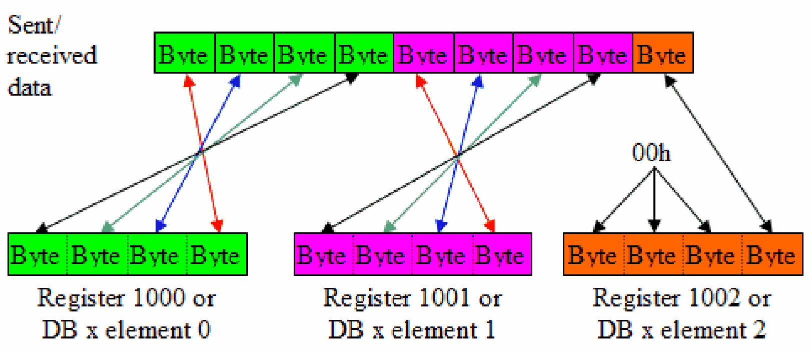
Example

There are 9 Bytes to send/receive.

➔ Without byte swapping (SendData / ReceiveData):


Sent/ received data
Byte Byte Byte Byte Byte Byte Byte Byte Byte
Byte Byte Byte Byte
Byte Byte Byte Byte
Byte Byte Byte Byte
Register 1000 or DB x element 0
Register 1001 or DB x element 1
Register 1002 or DB x element 2

➔ With byte swapping (SendDataRev / ReceiveDataRev):


Sent/ received data
Byte Byte Byte Byte Byte Byte Byte Byte Byte
00h
Byte Byte Byte Byte
Byte Byte Byte Byte
Byte Byte Byte Byte
Register 1000 or DB x element 0
Register 1001 or DB x element 1
Register 1002 or DB x element 2

Note:

If the buffer is a text, bytes are never swapped.

IP address decoding
The IP address can be given as a value in a text, register or constant. The IP address can also be the node in a register or constant value. A constant value can only be a node.

IP address in text:
In a text, the IP address is coded as text in the form of 4 decimal numbers separated by points, e.g. "192.168.12.14"

IP address in register:
Coded in a register, the IP address will be a node if the higher word is 0. If it is not 0, it will take the form of an address.

The IP address is coded as 4 hex numbers:

aa	bb	cc	dd

The IP address will have this format in reception/connection information.

Ex. 0C0A80C0Eh for the IP address 192.168.12.14

IP address in constant:
Coded in a constant, the IP address is always the node number.

See also
For more information, consult the manual Ether-S-Bus 26/776.

9.7 Profibus-DP

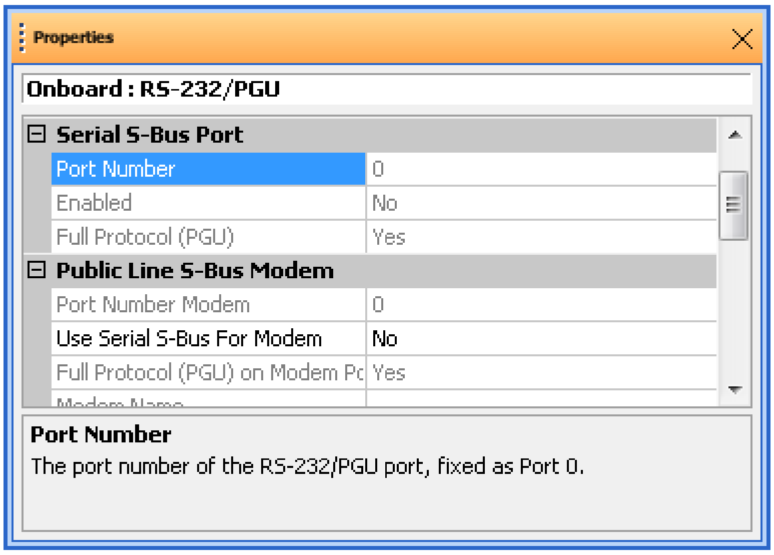
Profibus (PROcess FIeld BUS) is the standard industrial fieldbus.

For more information, search for **Profibus-DP** in the SBC website http://www.sbc-support.com.

9.8 Channel Number

The communication channel or port number is dependant on the PCD Device Type and onboard communications hardware.

Use the Device Configurator to see the available ports.
Configure the device type and the onboard communications. The "Properties" window displays the channel number required for your hardware.



Properties

Onboard : RS-232/PGU

Serial S-Bus Port	
Port Number	0
Enabled	No
Full Protocol (PGU)	Yes
Public Line S-Bus Modem	
Port Number Modem	0
Use Serial S-Bus For Modem	No
Full Protocol (PGU) on Modem Pc	Yes

Port Number

The port number of the RS-232/PGU port, fixed as Port 0.

9.9 SASI - Assign Serial Interface

Description

Processes a Text which contains the necessary information to initialize a communication channel:

The 1st operand is the serial channel number.

The 2nd operand is the number of a text which contains the channel operating definitions.

This initialization must be repeated for each channel to be used.
The SASI instructions are usually placed in the XOB 16.
Each channel can work in different modes and at different speeds.

Format

```
SASI     [=] channel    ;channel number
         [=] text       ;definition SASI text number
```

Example

```
SASI     0     ;initialize channel 0
         100   ;using definitions in Text 100
```

Flags

ACCU		Unchanged
Status Flags	E	The Error flag is set if the definition text is missing or invalid, if the channel does not exist or already defined as PGU channel, if the station number has not been defined. **Tip:** $SASI..$ENDSASI can be used to detect invalid SASI texts

See also

SASI Texts (Mode C)
SASI Texts (Mode D & MM4)
SASI Text (Serial-S-Bus)
SASI Text (Profi-S-Bus)
SASI Text (Ether-S-Bus)
SASI Text (Profibus-DP)
SASI Mode OFF
For SASI Text (Mode Profibus-FMS): consult the manual "PROFIBUS-FMS with SAIA PCD", ref. 26/742

Practical example
Initialize the channel 1 for text mode with a speed of 4800 Baud, 7 data bits, even parity and one stop bit. The SASI instruction is placed in XOB 16.

```
XOB     16      ;cold Start Exception Organization Block
SASI    1       ;assign serial channel 1
        10      ;with parameters in Text 10
EXOB

TEXT 10 "UART:4800,7,E,1;"
        "MODE:MC0;"
        "DIAG:F1000,R4000;"
```

The text could also be written on one line:

```
TEXT 10 "UART:4800,7,E,1;MODE:MC0;DIAG:F1000,R4000;"
```

9.10 SASII - Assign Serial Interface Indirect

Description
Processes a Text which contains the necessary information to initialize a communication channel.
This instruction works in the same way as the SASI instruction.
The difference is that it works in indirect mode.
Indirect mode means that the number of the channel and the definition text number can be given by the content of registers.

Note
This instruction cannot be used with Function Block parameters (= n).
Temporary Registers, defined with TEQU, cannot be used.

Format

```
SASII     channel     ;channel number or Register containing the number
          text_reg    ;Register containing the number of the SASI Text
```

The definition Text is the same as for the SASI instruction.

Example

```
SASII     1      ;initialize serial channel 1
          R 1    ;using the definition Text whose number is in R 1
```

Flags

ACCU		Unchanged
Status Flags	E	The Error flag is set if the definition text is missing or invalid, if the channel does not exist or already defined as PGU channel, if the station number has not been defined. **Tip:** $SASI..$ENDSASI can be used to detect invalid SASI texts

See also
SASI
SASI Texts (Mode C)
SASI Texts (Mode D & MM4)
SASI Text (Serial-S-Bus)
SASI Text (Profi-S-Bus)
SASI Text (Ether-S-Bus)
SASI Text (Profibus-DP)

9.11 SASI Text (Mode D & MM4)

Note: Mode D and MM4 are not supported by new PCD models.

Description
This is a special text definition for the SASI instruction. The contents depends of the communications mode.

Format

```
                          ;xxxx is a Text number
TEXT xxxx "<uart_def>;"   ;baud rate, data length, parity and stop bit
          "<mode_def>;"   ;communications mode
          "<diag_def>;"   ;diagnostic elements
```

Validating SASI texts
The $SASI and $ENDSASI directives can used to enclose SASI texts. This causes SASM to check the syntax of the text and all characters are converted in uppercases.

<uart_def>

Defines the baud rate, data length, parity, number of stop bits, timeout

Format: `"UART:<baudrate>,<char_len>,<parity>,<stop_bit>[,<timeout>];`

`<baud_rate>`	`<char_len>`	`<parity>`	`<stop_bit>`	`<time_out>`	or by default
110	7	E (even)	1	10..15000 ms	15 s
150	8	O (odd)	2		9 s
300		L (low)			5 s
600		H (high)			3 s
1.200		N (none)			2 s
2.400					1 s
4.800					0,5 s
9.600					0,25 s
19.200					0,2 s
38.400					0,1 s

Baud rate
Baudrates up to 38 400 or 115 200 bps are supported by all PCD modules regardless of hardware, firmware version.
(exception : 20mA current loop - only up to 9600 bps).
The baud rate 38'400 bps is not supported on the old PCD hardware.

When assigning an interface as 38.4 Kbps it should also be noted that, for physical reasons, some baud rates are no longer possible for assigning the second DUART interface.

For interfaces 0 + 1 (DUART 1) and 2 + 3 (DUART 2) respectively, the following combinations of baud rates are not possible :
38.4 Kbps + 38.4 Kbps
38.4 Kbps + 19.2 Kbps
38.4 Kbps + 150 bps

38.4 Kbps + 110 bps

If an attempt is still made to assign a prohibited combination, the error flag is set and XOB 13 is called.

CPU load for communications at 38.4 Kbps
Since S-Bus communication does not use a separate communications processor, data transmission at 38 400/115 200 bps makes corresponding demands on CPU capacity.
If the communications throughput is large, it can demand up to 40% of CPU capacity. This in turn means that processing of the user program is slowed down by the same factor.

<mode_def>

Defines the operating mode of the serial channel.

Format: `"MODE:<mode>,[<reg>];"`

`<mode>`	**Description**
MD0	Mode D Master
	Mode D Master via LAN1; Register = SCON status
SD0	Mode D Slave
	Mode D Master via LAN1; Register = SCON status
MM4	Mode MM4
OFF	De-assignation of the serial line. SASI Mode OFF

<reg>	D Mode with LAN 1 When using the SAIA PCDLAN 1, the PCA2.T9x interface uses a Register to inform the PCD about the status of the connection. For more information, see the SCON instruction.
	MM4 <mode_opt> consists of the following: <BCS_opt>,<trpartner>,<trinfo>,<repartner>,<reinfo>,<rechar>

`<mode_opt>`	**Value**	**Description**
`<BCS_opt>`	0 or 1	Block Check Sum (0: no BCS, 1: CRC16
`<trpartner>`	R xxxx	Transmission partner station number
`<trinfo>`	R xxxx	Remote ACK information
`<repartner>`	R xxxx	Reception partner station number
`<reinfo>`	R xxxx	Receive information
`<rechar>`	R xxxx	Number of received characters

<diag_def>

Defines the communications diagnostics media.

Format: `"DIAG:<dia_elem>,<dia_reg>;"`

	Type	**Description**

<dia_elem>	O xxxx F xxxx	Base address of 8 consecutive Flags (or Outputs)
<dia_reg>	R xxxx	Address of a register for diagnostic

where xxxx is a valid address

The 8 Flags give information about the status of the serial line. In case of error when executing a communication instruction, more information can be obtained by examining the contents of the Diagnostic Register.

Diagnostic Flags

The SASI texts DIAG address is the base address of 8 consecutive Outputs or Flags, used as follows:

Address	Name	Description
xxxx	RBSY	Receiver busy
xxxx+1	RFUL	Receive buffer full
xxxx+2	RDIA	Receiver diagnostic
xxxx+3	TBSY	Transmitter busy
xxxx+4	TFUL	Transmit buffer full
xxxx+5	TDIA	Transmitter diagnostic
xxxx+6	XBSY	Text Busy
xxxx+7	NEXE	Not executed

RBSY, Receiver Busy
RBSY is High when the receiver is busy.

RFUL: Receive Buffer Full
RFUL is High when a correct data frame has been received.

RDIA: Receiver Diagnostic
RDIA is set High if the PCD detects an error during reception of a character. See also the Diagnostic Register.
RDIA will be reset when all receiver diagnostic bits (0..15) in the Diagnostic Register are reset.

TBSY: Transmitter Busy
TBSY is set High when the PCD is transferring data.
TBSY is set Low when the telegram has been acknowledged or when the number of retries is reach.

TFUL: Transmit Buffer Full

D	Not used
M M4	TFUL is High when the acknowledgment has been received.

TDIA: Transmitter Diagnostic
TDIA is set High when the PCD detects an error during transmission of a character. See also the Diagnostic Register.
TDIA will be reset when all transmitter diagnostic bits (16..31) in the diagnostic register are reset.

XBSY: Text busy

D	XBSY is High when a connection via the LAN1 is open.
MM4	XBSY is High when there is activity on the LAC network (STXM instruction)

NEXE: Not executed
If the PCD is unable to perform the requested operation, NEXE is set High. See also the Diagnostic Register.

Diagnostic Register

If the diagnostic flag TDIA or RDIA is high, see the Diagnostic Register for details.
Any bit which has been set high in the Diagnostic Register remains High until manually reset by the user program or the debugger.

Bit	Description	Cause	Mode				
			C	D	SBUS	MM4	PROFI-BUS
0	Overrun error	Should never occur (notify SAIA)	x	x	x	x	
1	Parity error	Received a character with a parity error	x	x		x	
2	Framing error	Usually caused by an incorrect baud rate	x	x	x	x	
3	Break	Break in data line	x	x	x	x	
4	BCC error	Bad Block Check Code (or CRC16)		x	x	x	
5	S-Bus PGU status	S-Bus PGU with Public Line modems			x		
6	End of transmit	Transmission ended SASI OFF		x	x		
7	Overflow error	Receive buffer overflow	x	x		x	
8	Length error	The telegram length is invalid			x		
9	Format error	Invalid telegram format		x		x	x
10	Address error	Adress of ACK is invalid			x	x	
11	Status error	PCD in false status			x		
12	Range error	Invalid element address		x	x		x
13	Value error	Error in the received value			x		
14	Missing media err	Address of media not defined or invalid			x		
15	Program error	Read from an empty receive buffer	x				
		LAN 1 not assigned or invalid station nb		x	x		
16	Retry count	Indicates the number of retries (in binary)			x		
17							
18	Transmission off	Sending is suspended (CTS = L or XOFF)	x				
19							
20	NAK response	NAK was received		x	x	x	x
21	No response	No response was received after		x	x	x	

		timeout					
22	Multiple NAK	NAK received after retries		x	x	x	
23	TX buffer full	No more space in transmit buffer	x				
	TS Delay	No CTS after the TS Delay			x		
24	Enquiry error	No response to ENQ after retries		x			
25	Format error	Invalid definition text	x				
		Invalid command		x			x
26	Partner error	A problem has occured with the partner				x	
27	Network error	A problem has occured on the network				x	
28	Range error	Invalid element address		x	x	x	x
29							
30	Receive error	Error occurred		x			x
31	Program error	Attempt to transmit when unauthorized		x	x	x	x

Examples of SASI Texts

Mode MD0 (Master)

```
$SASI
TEXT 40     "UART:9600,7,E,1,3000;"
            "MODE:MD0,R1;"
            "DIAG:F1000,R4000;"
$ENDSASI
```

Register R 1 is used to store the connection state of the LAN1.

Mode SD0 (Slave)

```
$SASI
TEXT 30     "UART:9600,7,E,1;MODE:SD0;DIAG:F1000,R4000;"
$ENDSASI
```

Mode MM4

```
$SASI
TEXT 50     "UART:9600,8,N,1,300;"
            "MODE:MM4,0,R100,R101,R102,R103,R104;"
            "DIAG:F1000,R1000;"
$ENDSASI
```

9.12 SASI Text (Mode C)

Description
This is a special text definition for the SASI instruction. The contents depends of the communications mode.

Format

```
                           ;xxxx is a Text number
```

```
TEXT xxxx    "<uart_def>;"   ;baud rate, data length, parity and stop bit
             "<mode_def>;"   ;communications mode
             "<diag_def>;"   ;diagnostic elements
            ["<rx_buf>;"]    ;optional receive buffer length (default = 1)
            ["<tx_buf>;"]    ;optional transmit buffer length
```

Validating SASI texts

The $SASI and $ENDSASI directives can used to enclose SASI texts. This causes S-Asm to check the syntax of the text and all characters are converted to uppercase.

<uart_def>

Defines the baud rate, data length, parity, number of stop bits, timeout

Format: `"UART:<baudrate>,<char_len>,<parity>,<stop_bit>;`

<baud_rate>	<char_len>	<parity>	<stop_bit>
110	7	E (even)	1
150	8	O (odd)	2
300		L (low)	
600		H (high)	
1.200		N (none)	
2.400			
4.800			
9.600			
19.200			
38.400			
57.600			
115 200			

Baud rate

Baudrates up to 38 400 or 115 200 bps are supported by all PCD modules regardless of hardware, firmware version.
(exception : 20mA current loop - only up to 9600 bps).
The baud rate 38 400 bps is not supported on the old PCD hardware.

When assigning an interface as 38.4 Kbps it should also be noted that, for physical reasons, some baud rates are no longer possible for assigning the second DUART interface.

For interfaces 0 + 1 (DUART 1) and 2 + 3 (DUART 2) respectively, the following combinations of baud rates are not possible :
38.4 Kbps + 38.4 Kbps
38.4 Kbps + 19.2 Kbps
38.4 Kbps + 150 bps
38.4 Kbps + 110 bps

If an attempt is still made to assign a prohibited combination, the error flag is set and XOB 13 is called.

CPU load for communications at 38.4 Kbps

Since S-Bus communication does not use a separate communications processor, data transmission at 38 400/115 200 bps makes corresponding demands on CPU capacity.
If the communications throughput is large, it can demand up to 40 % of CPU capacity. This in turn

means that processing of the user program is slowed down by the same factor.

<mode_def>

Defines the operating mode of the serial channel.

Format: `"MODE:<mode>;"`

`<mode>`	**Description**		
MC0	Mode C without automatic handshaking. The user must control by himself the control signals with the SICL and SOCL instructions.		
MC1	Mode C using RTS and CTS handshaking. The RTS control signal is automatically positioned by the PCD in function of the remaining space in the reception buffer. The CTS signal influences the transmission of the		
	RTS	Low High	Receive buffer contains more than 450 characters Receive buffer contains less than 300 characters
	CTS	Low High	Transmission is stopped Transmission is resumed
MC2	Mode C with Xon/Xoff protocol. This mode is similar to the RTS/CTS handshaking and is used when no control signals are present (eg. current loop). Two special characters Xon (CTRL/Q) and Xoff (CTRL/S) are sent to control the transmission of the partner		
	Receiver send when	Xoff Xon	Receive buffer contains more than 450 characters Receive buffer contains less than 300 characters
	Transmitter receives then	Low High	Transmission is stopped Transmission is resumed
MC3	Mode C with echo. This mode is used when communicating with a terminal. All received characters are sent back to the terminal screen.		
MC4	Mode C for RS485 interface The MC4/MC5 modes are low level modes which will set the RS485 driver/receiver in drive mode only during the transmission of information (character/text) and will be set by default to receive at any other time. However the XBSY and TBSY flags don't work in the same way. For MC4: • The reset to receipt mode happens between 0 and 1 ms after the end of the last character sent. • The flags XBSY/TBSY are reset to 0 between 0 and 1 ms after the end of the last character sent. For MC5: • The reset to receipt mode happens one bit time (time needed to transfer 1 bit) after the end of the last character sent. • The flags XBSY/TBSY are reset to 0 between 0 and 1 ms after the begin of the last character sent.		
OFF	De-assignation of the channel. SASI Mode OFF		

<diag_def>

Defines the communications diagnostics elements.

Format: `"DIAG:<dia_elem>,<dia_reg>;"`

	Type	Description
`<dia_elem>`	O xxxx F xxxx	Base address of 8 consecutive Flags (or Outputs)
`<dia_reg>`	R xxxx	Address of a register for diagnostic

where xxxx is a valid address

The 8 Flags give information about the status of the serial line. In case of error when executing an serial communication instruction, more information can be obtained by examining the contents of the Diagnostic Register.

Diagnostic Flags

The SASI texts DIAG address is the base address of 8 consecutive Outputs or Flags, used as follows:

Address	Name	Description
xxxx	RBSY	Receiver busy
xxxx+1	RFUL	Receive buffer full
xxxx+2	RDIA	Receiver diagnostic
xxxx+3	TBSY	Transmitter busy
xxxx+4	TFUL	Transmit buffer full
xxxx+5	TDIA	Transmitter diagnostic
xxxx+6	XBSY	Text Busy
xxxx+7	NEXE	Not used

RBSY, Receiver Busy
RBSY is High when at least one character is available in the reception buffer.
When all characters waiting in the reception buffer have been read with the SRXD instruction RBSY is cleared.

RFUL: Receive Buffer Full
RFUL is set High when the number of incoming characters in the PCD Receive buffer is equal to or greater than the value of rx_buf (Receive buffer length).
RFUL is Low when the number of characters remaining in the receive buffer is less than the vale of rx_buf. The internal reception buffer of the PCD always has room for 512 characters.

RDIA: Receiver Diagnostic
RDIA is set High if the PCD detects an error during reception of a character. See also the Diagnostic Register.
After execution of a communication instruction, RDIA is reset only if all receiver diagnostic bits (0...15) in the diagnostic register are 0.

TBSY: Transmitter Busy
TBSY is set High when the PCD transmits characters over the serial line. TBSY is set Low when all

characters from the Transmission buffer have been transmitted

TFUL: Transmit Buffer Full
TFUL is set High when the number of characters remaining in the PCD transmission buffer is greater than or equal to the value declared for tx_buf (Transmit buffer length).
The TFUL is reset when the number of characters remaining in the Transmit buffer is less than the value of TBUF.

TDIA: Transmitter Diagnostic
TDIA is set High when the PCD detects an error during transmission of a character. See also the Diagnostic Register.
After execution of a communication instruction, TDIA is reset only if all transmitter diagnostic bits (16...31) in the Diagnostic Register are 0.

XBSY: Text busy
XBSY is set High when the PCD transmits a text (STXT); when all the text has been transmitted XBSY is reset. Note: XBSY is reset at the beginning of the sending of the last character.

Diagnostic Register
If the diagnostic flag TDIA or RDIA is high, the diagnostic register content will help you to found the communication trouble.
Any bit which has been set high in the diagnostic register remains so, until manually reset by the user program or the debugger.

	Bit	**Description**	**Cause**
RECEIVER	0	Overrun error	Should never occur (notify Saia Burgess Controls)
	1	Parity error	Received a character with a parity error
	2	Framing error	Usually caused by an incorrect baud rate
	3	Break	Break in data line
	4		
	5		
	6		
	7	Overflow error	Receive buffer overflow
	8		
	9		
	10		
	11		
	12		
	13		
	14		
	15	Program error	Read from an empty receive buffer
T	16		
	17		
	18	Transmission off	Sending is suspended (CTS = L or XOFF)
	19		
	20		
	21		
	22		

RANSMITTER	23	TX buffer full	No more space in transmit buffer
	24		
	25	Format error	Invalid definition text
	26		
	27		
	28		
	29		
	30		
	31		

Bit 0: Overrun Error
Set high when there is an overrun of the internal buffer of the DUART.
Cause : Baud rate assigned is too high, the CPU can no longer process all characters received.

This can happen if one CPU is involved in communications requiring a high rate of data transmission via several interfaces simultaneously. It is theoretically possible for all interfaces of a CPU (excluding the 20mA current loop) to be assigned the maximum Baud rate of 19.200 bps at the same time. In practice, however, this error can arise when there is a very high level of communication over several interfaces. The system program handles the interfaces with differing priorities. The highest priority is allocated to interface 0, declining to interface 3.

Remedy :
- Reduce Baud rate.
- For fast communication, use an interface with high priority, if possible.

Bit 2: Framing Error
Set high when a character is received with a framing error (missing stopbit). This is usually caused by setting the Baud rate wrongly.

Bit 3: Break Error
Set high when an interruption is noticed during receipt of a character.
Cause : Data line broken or wrongly set Baud rate.

Bit 15: Program Error
Set high during execution of a SASI instruction with the definition SS1 mode, if the user program header has not been configured for the S-Bus slave station, or if the configuration is invalid.

<rx_buf>

Format: `"RBUF:<rbuf_len>;"`

Defines the communication reception buffer limit.
The <rx_buf> is are only used for mode C

	Value	**Description**
<rbuf_len>	1.. 511	Receive buffer length

The Receive Buffer has always space for 512 x 8bit characters.
The RBUF definition (1511) indicates when to set the Receive Buffer Full status (RFUL).

If no indication is given, the default value 1 will be taken, i.e. after only one received character the RFUL flag will be set.

<tx_buf>
Defines the communication transmission buffer limit.

Format: `"TBUF:<tbuf_len>;”`

The <tx_buf> is only used for mode C

	Value	**Description**
<tbuf_len>	1.. 511	Transmission buffer length

Similar to the Receive Buffer.
The TBUF definition (1511) indicates when to set the Transmit Buffer Full status (TFUL).

If no indication is given, the default value 1 will be taken.

Examples of SASI Texts

Mode MC0

```
    $SASI
    TEXT 10     "UART:9600,7,E,1;MODE:MC0;DIAG:F1000,R4000;"
    $ENDSASI

    $SASI
    TEXT 11     "UART:19200,8,E,2;"
                "MODE:MC0;"
                "DIAG:F0123,R4000;"
                "RBUF:128;"
                "TBUF:32;"
    $ENDSASI
```

Mode MC2

```
    $SASI
    TEXT 20     "UART:4800,8,N,1;MODE;MC2;DIAG:F0,R100;"
                "RBUF:25;"
    $ENDSASI
```

See also
Using Symbols in $SASI Texts

9.13 SASI Text (Serial S-Bus)

Description
The master and slave station number must be configured from device configurator and completed with a special text definition for the SASI instruction.
The master and slave channel settings are fully configured from SASI text.

Format

```
                            ;xxxx is a Text number
TEXT xxxx  "<UART_DEF>;"  ;baud rate, timeout, TS-Delay, TN-Delay, break len
           "<MODE_DEF>;"  ;comms mode and register for slave station number
           "<DIAG_DEF>;"  ;diagnostic R and F for the serial communication
```

Validating SASI texts
The $SASI and $ENDSASI directives can used to enclose SASI texts. This causes S-Asm to check the syntax of the text and all characters are converted to uppercase.

<UART_DEF>

Defines Baud rate, Timeout, TS-Delay, TN-Delay and Break-Length.

The definitions of character length, parity and stop bits are not required, as the S-Bus protocol includes the following definitions as fixed settings :

Character length	8 bits	
Stop bit	1 bit	
Parity bit	Mode SM2/SS2	Data mode
	Mode SM1/SS1	Parity bit "1" for address character
		Parity bit "0" for data character
	Mode SM0/SS0	With Break character

Format: `"UART:<Baudrate>[,<Timeout>][,TS-Delay>][,TN-Delay][,Break-Length];"`

TimeOut, TS-Delay and TN-Delay are optional and normally only needed to be defined when a modem is used.

Baudrate		**[Timeout]**		**[TS-Delay]**	**[TN-Delay]**		**[Break-Length]**
	adjustable	or default value			adjustable	or default	
		Saia PCD-NT system	Other PCD			value	adjustable
110	1…15000 ms		15000 ms	1…15000 ms	1…15000 ms	27 ms	4…25 characters
150			9000 ms			20 ms	
300			5000 ms			20 ms	
600			3000 ms			5 ms	
1200		2000 ms	2000 ms			3 ms	
2400		1000 ms	1000 ms			2 ms	
4800		500 ms	500 ms			2 ms	
9600		250 ms	250 ms			1 ms	
19200		200 ms	200 ms			1 ms	
38400		200 ms	100 ms			1 ms	
57.600		200 ms					
115 200		200 ms					

Baud rate
Baudrates up to 38 400 or 115 200 bps are supported by all PCD modules regardless of hardware, firmware version.
(exception : 20mA current loop - only up to 9600 bps).
The baud rate 38 400 bps is not supported on the old PCD hardware.

When assigning an interface as 38.4 Kbps it should also be noted that, for physical reasons, some baud rates are no longer possible for assigning the second DUART interface.

For interfaces 0 + 1 (DUART 1) and 2 + 3 (DUART 2) respectively, the following combinations of baud rates are not possible :
38.4 Kbps + 38.4 Kbps
38.4 Kbps + 19.2 Kbps
38.4 Kbps + 150 bps
38.4 Kbps + 110 bps

If an attempt is still made to assign a prohibited combination, the error flag is set and XOB 13 is called.

CPU load for communications at 38.4 Kbps
Since S-Bus communication does not use a separate communications processor, data transmission at 38 400/115 200 bps makes corresponding demands on CPU capacity.
If the communications throughput is large, it can demand up to 40 % of CPU capacity. This in turn means that processing of the user program is slowed down by the same factor.

Timeout
This value defines the maximum time after sending a read telegram (instruction SRXM), during which the reply telegram must be received from the station addressed.
If no valid reply is received within this time, the last telegram transmitted is repeated and the corresponding diagnostic elements are set. Two repeat transmissions are the maximum for any telegram.

TN-Delay (Delay time on turnaround)
This parameter defines the delay time before the RTS signal is switched on at the RS 232 and RS 422 interfaces, or before the transmitter is
switched on at the RS 485 interface. A telegram is sent at the earliest after this delay time has elapsed.

TS-Delay (Training Sequence Delay time)
This parameter defines a monitoring time for the CTS (Clear To Send) signal of a connected device.
The PCD sends a telegram as soon as the connected device (modem) has shown its readiness to receive by setting the CTS signal, or at the end of the TS-Delay time.
If the CTS signal has not been set by the end of the TS-Delay time, bit 23 (CTS-Timeout) is set in the diagnostic register.
Monitoring and handling of the CTS signal is only active if the parameter has been defined in the SASI text. Otherwise the CTS signal is ignored.
The standard value for the TS-Delay time is 0 ms.
If, within the timeout defined by the SASI instruction, the master station receives an incomplete or invalid reply telegram, the telegram sent before is transmitted again.

Break-Length
This parameter allows the length of the break signal to be adjusted in SM0 mode. This is used to differentiate between data and address characters.
An address character is identified by a preceding break signal. A break signal is only sent by the master station in SM0 mode and can therefore also only be adjusted from that station.
It is not normally necessary to change the break length.

Break signal : Data line = low for duration of n characters including stop bit.

<MODE_DEF>

Defines communications mode and a register for the slave station number.

Format : `"MODE:<sbus_mode>[,<dest_reg>[,<secure_mode>]];"`

<sbus_mode>	**Description**
SM2	Serial S-Bus master, Data Mode
SM1	Serial S-Bus master, with parity bit control
SM0	Serial S-Bus master, with break character
SS2	Serial S-Bus slave, Data Mode
SS1	Serial S-Bus slave, with parity bit control
SS0	Serial S-Bus slave, with break character
GS2	Serial S-Bus Gateway slave, Data Mode
GS1	Serial S-Bus Gateway slave, with parity bit control
GS0	Serial S-Bus Gateway slave, with break character
GM	Serial S-Bus Gateway master
OFF	De-initialize the serial line (see also SASI Mode OFF)

<dest_reg>	Master: This Register defines the address of the remote station. Slave: Not used

<secure_mode>	For SM2 and GM mode only, see note below 0 = turn off secure data mode 1 = turn on secured data mode Default is 1

Note: "Secure data mode" is a updated S-Bus protocol which assigns sequence numbers to each S-Bus telegram, for better communications reliability. If the slave device does not support secure data mode then it can be turned off with this option.

The remote station address has to be loaded into the register before executing SRXM/STXM.

Register address field (32 bit)			
More significant word		Less significant word	
			S-Bus-address

Example:

```
LD     R 100    ;Remote address register
       10       ;S-Bus address 10
STXM   10       ;channel no. 10
       100      ;Transmit 100 elements
       F 500    ;Flag 500 ...599
       O 32     ;to outputs 32..131
```

<DIAG_DEF>

Defines diagnostic elements for Serial-S-Bus communication.

Format: `"DIAG:<diag_elem>,<diag_reg>;"`

	Type	Description
<diag_elem>	F xxxx O xxxx	Base address of 8 consecutive Flags (or Outputs)
<diag_reg>	R xxxx	Address of a register for diagnostic

The 8 Flags give information about the status of the serial line. In case of error when executing a communication instruction, more information can be obtained by examining the contents of the Diagnostic Register.

Diagnostic Flags

The SASI texts DIAG address is the base address of 8 consecutive Outputs or Flags, used as follows:

Address	Name	Description
xxxx	RBSY	Receiver busy
xxxx+1	RFUL	Receive buffer full
xxxx+2	RDIA	Receiver diagnostic
xxxx+3	TBSY	Transmitter busy
xxxx+4	TFUL	Not used
xxxx+5	TDIA	Transmitter diagnostic
xxxx+6	XBSY	SASI permission
xxxx+7	NEXE	Not executed

Receiver Busy (RBSY)
Set high when a slave station receives a telegram. The flag is reset as soon as the reply telegram has been sent. This flag has no significance in the case of the master station

Receive Buffer Full (RFUL)
Set high when elements in the slave station have been changed by the master station.

Receiver Diagnostic (RDIA)
Set high when an error is noticed during receipt of a telegram. A detailed description of the error can be obtained from the Diagnostic Register (bits 0..15).
After execution of a communication instruction, RDIA is reset only if all receiver diagnostic bits (0...15) in the diagnostic register are 0.

Transmitter Busy (TBSY)
Set high while transmission is taking place.
Master station :
It is set high during execution of an STXM or SRXM instruction. The flag is reset as soon as a valid reply is received.
Slave station :
It is set high while the reply is transmitted.

Transmitter Diagnostic (TDIA)

Set high if an error occurs during transmission of a telegram. A detailed description of the error can be obtained from the Diagnostic Register.
After execution of a communication instruction, TDIA is reset only if all transmitter diagnostic bits (16...31) in the diagnostic register are 0.

Interface busy (XBSY)
Low when the user has the permission to perform a SASI OFF to undo the S-Bus PGU for Public Line modem.

Not Executed (NEXE)
Set high if an instruction (STXM or SRXM) has not been completed after three attempts. The flag is reset by the next S-Bus instruction.

Diagnostic Register

If the diagnostic flag TDIA or RDIA is high, the diagnostic register content will help you to found the communication trouble.
Any bit which has been set high in the diagnostic register remains so, until manually reset by the user program or the debugger.

	Bit	**Designation**	**Description**
	0	Overrun error	Overrun of the internal receiver buffer
	1		
	2	Framing error	Usually caused by an incorrect baud rate
	3	Break error	Break in data line (No signification in mode SM0/SS0)
R	4	BCC error	Bad Block Check Code or CRC-16
E	5	S-Bus PGU status	S-Bus PGU with Public Line modems
C	6	SASI OFF permission	SASI OFF permission
E	7		
I	8	Length error	The telegram length is invalid
V	9		
E	10	Address error	Address of ACK is invalid
R	11	Status error	PCD in false status, cannot execute command
	12	Range error	Invalid element address
	13	Value error	Error in the received value
	14	Missing media error	Address of media not defined or invalid
	15	Program error	Station number not allocated (or invalid)
	16	Retry count	Indicates the number of retries (in binary)
	17		(telegram repeats in binary representation)
T	18		
R	19		
A	20	NAK response	Negative response (NAK) was received
N	21	Missing response	No response was received after timeout
S	22	Multiple NAK	NAK received after retries
M	23	CTS-Timeout	No CTS set after TS delay
I	24		
T	25		
T	26		

E	27		
R	28	Range error	Invalid element address
	29		
	30		
	31	Program error	Attempt to transmit when unauthorised

Bit 0: Overrun Error
Set high when there is an overrun of the internal buffer of the DUART.
Cause : Baud rate assigned is too high ? the CPU can no longer process all characters received.
This can happen if one CPU is involved in communications requiring a high rate of data transmission via several interfaces simultaneously.
It is theoretically possible for all interfaces of a CPU (excluding the 20mA current loop) to be assigned the maximum Baud rate of 19.200 bps at the same time.
In practice, however, this error can arise when there is a very high level of communication over several interfaces.
The system program handles the interfaces with differing priorities. The highest priority is allocated to interface 0, declining to interface 3.
Remedy :
- Reduce Baud rate.
- For fast communication, use an interface with high priority, if possible.

Bit 2: Framing Error
Set high when a character is received with a framing error (missing stopbit). This is usually caused by setting the Baud rate wrongly.

Bit 3: Break Error
Set high when an interruption is noticed during receipt of a character.
Cause : Data line broken or wrongly set Baud rate.

Bit 4: BCC or CRC-16 Error
Set high if a CRC-16 error is identified on the incoming telegram. The incoming telegram is rejected.
Reaction of Slave : The received telegram will be ignored
Master : The received telegram will be ignored and the last telegram will be retransmitted.
Cause : Interference on the data line.
Remedy : Check electrical installation.

Bit 5: S-Bus PGU Status
Shows the current S-Bus PGU with Public Line Modem (PLM)
"1": S-Bus port is in STANDBY status , waiting for modem connection.
"0": No S-Bus PGU PLM port configured or in FINAL status (PCD ready in mode S-Bus level 2 for modem or S-Bus PGU PLM undone yet.

Bit 6: SASI OFF Permission
Indicates that somebody has disabled an UNDO/REDO process of the S-Bus PGU PLM in performing a RUN or STOP via S-Bus or PG4/PG3 Utilities during the SASI OFF execution delay period.

Bit 8: Length Error
Set high when a telegram is received with invalid length. This error cannot arise in a network made up exclusively of PCD stations.
The error indicates that an invalid telegram has been received from an external system. This results in a NAK response.

Bit 10: Address Error
Set high if an invalid telegram is received (incorrect command code).

Cause : Same as for Length Error (there is no NAK response).

Bit 11: Status Error
Set high when the PCD can not execute a command request because the slave PCD is not in the correct status (Run/ Halt/Stop/Diconnected/…). Only used for S-Bus level 2

Bit 12: Range Error
Set high if an incoming telegram contains an invalid PCD element address. This error cannot arise in a network made up exclusively of PCD stations, as the master PCD monitors the element address range of telegrams as they are transmitted.
The slave station responds to this error with NAK.

Bit 13: Value Error
Set high when an invalid data value is received.
Example :
The STXM instruction is used in an attempt to load the clock. The value received for the hour is 30. However, the maximum range for the hour is only 0..23.
The slave station responds to this error with NAK.

Bit 14: Missing Media Error
Set high when the addressed media is not defined or invalid media code for current request. Only used for S-Bus level 2

Bit 15: Program Error
Set high during execution of a SASI instruction with the definition SS1 mode, if the user program header has not been configured for the S-Bus slave station, or if the configuration is invalid.

Bits 16 and 17: Retry Count
Shows the number of repeat telegrams sent during execution of a SRXM or STXM instruction, represented in binary. Bit 16 is the LS bit. The quality of an S-Bus network can be judged by monitoring these two bits.

Bit 20: Negative Response
Set high if a NAK response is received from a slave. This means that the master has previously sent an invalid telegram. Check for the following errors: Value Error, Range Error and Length Error.

Bit 21: Missing Response
Set high if no response has been received from the slave station after the time-out has elapsed.
In this case, the telegram is retransmitted (maximum two times).
Possible causes :
- The slave station addressed does not exist.
- Installation error in network (wiring).
- The slave station has received a confused telegram with a CRC-16 error.

Remedies :
- Check slave station (connections, station number)
- Have the correct line termination and pull-up/down resistors been connected on the bus line at the first and last stations ?

Bit 22: Multiple NAK
Set high if, instead of the expected ACK or NAK, a different response is received from a slave station.
Possible causes :
- More than one slave with the same station number.
- More than one master in the network.
- Interference on the bus line.

Remedies :
- As for Missing Response error

Bit 23: CTS Timeout
Set high if the time between setting the control line RTS (by the PCD) and receiving the CTS (from the modem) exceeds the "TS Delay".

Bit 28: Range Error
Set high if the SRXM or STXM instructions indicate an element address (source or destination address) lying outside the permitted range.
Cause : Error in user program
Ranges monitored :

Inputs/Outputs	0..8191
Flags	0..8191
Timers/Counters	0..1599
Registers	0..8191

Example : During execution of the following STXM instruction, the Range Error bit is set high.

```
STXM   1         ;channel 1
       25        ;25 registers
       R 1000    ;base address source
       R 8172    ;base address destination
```

An attempt is made to transmit the contents of registers 1000 to 1024 in the master station to registers 4072 to 4096 in the slave station.

Bit 31: Program Error
Set high during execution of an STXM or SRXM instruction if the interface has been assigned in SS1 mode, or if a similar instruction is already executing (TBSY flag was not polled before executing the instruction).

Examples of SASI Texts

Mode S-Bus Parity mode (Master)

```
$SASI
TEXT 60   "UART:9600;MODE:SM1,R555;DIAG:F8000,R4005;"
$ENDSASI
```

Mode S-Bus Paritiy mode (Slave)

```
$SASI
TEXT 60   "UART:9600;MODE:SS1;DIAG:F8000,R4005;"
$ENDSASI
```

Mode S-Bus Data mode(Slave)

```
$SASI
TEXT 60   "UART:9600;MODE:SS2,R55;DIAG:F8000,R4005;"
$ENDSASI
```

See also
Using Symbols in Texts

9.14 SASI Text (Profi-S-Bus)

Description
The master and slave channel settings must be configured from device configurator and completed with a special text definition for the SASI instruction:

- The master SASI text defines the diagnostic flags, register address and options necessary to communication instructions: STXM/SRXM/...
- The slave SASI text is not necessary if the user don't need the diagnostic flags and registers. All definitions necessary are already present in device configurator.

Format

```
                               ;xxxx is a Text number
TEXT xxxx    "<MODE_DEF>;"     ;comms mode and Register for slave station number
             "<DIAG_DEF>;"     ;diagnostic elements for serial communication
            ["<SAP_DEF>;"]     ;master option to define the FDL sap number
            ["<TOUT_DEF>;"]    ;master option to define the Timeout value
```

Validating SASI texts
The $SASI and $ENDSASI directives can used to enclose SASI texts. This causes S-Asm to check the syntax of the text and all characters are converted to uppercase.

<MODE_DEF>

Defines communications mode and a register for the slave station number.

Format : `"MODE:<sbus_mode>[,<dest_reg>];"`

<sbus_mode>	Description
PSM	Mode Profi-S-Bus Master
PSS	PSS Mode Profi-S-Bus Slave

<dest_reg>	Master: this register defines the address of the remote station. Slave: do not defined

The remote station address has to be stored in the register before to send the commands STXM/SRXM.
This addressing uses two address fields, the upper and the lower part of the address register.

Register address field (32 bit)			
More significant word		Less significant word	
Not used	FDL/Profibus address	Not used	S-Bus-address

Example:

```
LDL     R 100     ;Remote address register
        10        ;S-Bus address 10
LDH     R 100
        15        ;FDL/Profibus address 15
STXM    10        ;channel no. 10
        100       ;Transmit 100 elements
        F 500     ;Flag 500 ...599
        O 32      ;to outputs 32..131
```

<DIAG_DEF>

Defines diagnostic elements for Profi S-Bus communication.

Format: `"DIAG:<diag_elem>,<diag_reg>;"`

	Type	**Description**
<diag_elem>	F xxxx O xxxx	Base address of 8 consecutive Flags (or Outputs)
<diag_reg>	R xxxx	Address of a register for diagnostic

The 8 Flags give information about the status of the serial line. In case of error when executing a communication instruction, more information can be obtained by examining the contents of the Diagnostic Register.

Diagnostic Flags

The SASI texts DIAG address is the base address of 8 consecutive Outputs or Flags, used as follows:

Address	**Name**	**Description**
xxxx	RBSY	Receiver busy
xxxx+1	RFUL	Receive buffer full
xxxx+2	RDIA	Receiver diagnostic
xxxx+3	TBSY	Transmitter busy
xxxx+4	TFUL	Transmitter full
xxxx+5	TDIA	Transmitter diagnostic
xxxx+6	XBSY	SASI permission
xxxx+7	NEXE	Not executed

Receiver Busy (RBSY)
Set high when a slave station receives a telegram. The flag is reset as soon as the reply telegram has been sent. This flag has no significance in the case of the master station

Receive Buffer Full (RFUL)
Set high when elements in the slave station have been changed by the master station.

Receiver Diagnostic (RDIA)
Set high when an error is noticed during receipt of a telegram. More information can be obtained from the Diagnostic Register (bits 0..15).
After execution of a communication instruction, RDIA is reset only if all receiver diagnostic bits (0…15) in the diagnostic register are 0.

Transmitter Busy (TBSY)
Set high while transmission is taking place.
Master station :
It is set high during execution of an STXM or SRXM instruction. The flag is reset as soon as a valid reply is received.
Slave station :

It is set high while the reply is transmitted.

Transmitter Diagnostic (TDIA)
Set high if an error occurs during transmission of a telegram. A detailed description of the error can be obtained from the Diagnostic Register.
After execution of a communication instruction, TDIA is reset only if all transmitterr diagnostic bits (16...31) in the diagnostic register are 0.

Interface busy (XBSY)
Low when the user has the permission to perform a SASI OFF to undo the S-Bus PGU for Public Line modem.

Not Executed (NEXE)
Set high if an instruction (STXM or SRXM) has not been completed after three attempts. The flag is reset by the next S-Bus instruction.

Diagnostic Register

If the diagnostic flag TDIA or RDIA is high, the diagnostic register content will help you to found the communication trouble.
Any bit which has been set high in the diagnostic register remains so, until manually reset by the user program or the debugger.

	Bit	**Designation**	**Description**
	0		
	1		
	2		
	3		
R	4		
E	5		
C	6		
E	7		
I	8	Length error	The telegram length is invalid
V	9		
E	10	Address error	Address of ACK is invalid
R	11		
	12	Range error	Invalid element address
	13	Value error	Error in the received value
	14		
	15		
	16	Retry count	Indicates the number of retries (in binary)
	17		(telegram repeats in binary representation)
T	18		
R	19		
A	20	NAK response	Negative response (NAK) was received
N	21	Missing response	No response was received after timeout
S	22	Multiple NAK	NAK received after retries
M	23		
I	24	FDL No ACK	No ACK over the FDL

T	25	FDL Negative ACK	Negative ACK over the FDL
T	26	FDL No Resource	No FDL resource in the partner station
E	27	FDL No Connection	No free SAP to open a connection to the server
R	28	Range error	Invalid element address
	29	TxNode_error	Node does not exist
	30		
	31	Program error	Attempt to transmit when unauthorised

Bit 8: Length Error
Set high when a telegram is received with invalid length. This error cannot arise in a network made up exclusively of PCD stations.
The error indicates that an invalid telegram has been received from an external system. This results in a NAK response.

Bit 10: Address Error
Set high if an invalid telegram is received (incorrect command code).
Cause : Same as for Length Error (there is no NAK response).

Bit 12: Range Error
Set high if an incoming telegram contains an invalid PCD element address.
This error cannot arise in a network made up exclusively of PCD stations, as the master PCD monitors the element address range of telegrams as they are transmitted.
The slave station responds to this error with NAK.

Bit 13: Value Error
Set high when an invalid data value is received.
Example:
The STXM instruction is used in an attempt to load the clock. The value received for the hour is 30. However, the maximum range for the hour is only 0..23.
The slave station responds to this error with NAK.

Bits 16 and 19: Retry Count
Shows the number of repeat telegrams sent during execution of a SRXM or STXM instruction, represented in binary. Bit 16 is the LS bit.
The quality of an S-Bus network can be judged by monitoring these two bits.

Bit 20: Negative Response
Set high if a NAK response is received from a slave. This means that the master has previously sent an invalid telegram.
Check for the following errors: Value Error, Range Error and Length Error.

Bit 21: Missing Response
Set high if no response has been received from the slave station after the time-out has elapsed.
In this case, the telegram is retransmitted (maximum two times).
Possible causes :
- The slave station addressed does not exist.
- Installation error in network (wiring).
- The slave station has received a confused telegram with a CRC-16 error.

Remedies :
- Check slave station (connections, station number)
- Have the correct line termination and pull-up/down resistors been connected on the bus line at the first and last stations ?

Bit 22: Multiple NAK
Set high if, instead of the expected ACK or NAK, a different response is received from a slave station.
Possible causes :
- More than one slave with the same station number.
- More than one master in the network.
- Interference on the bus line.

Remedies :
- As for Missing Response error

Bit 24: FDL No ACK
Set high if, no ACK is received over the FDL layer.

Bit 25: FDL Neg ACK
Set high if, negative ACK is received over the FDL layer.

Bit 26: FDL No Resource
Set high if there are no FDL resources in the partner station.
Possible causes:
- Partner is not present
- Too many clients access to the server
- Partner has no Profi-S-Bus Slave port open

Bit 27: FDL No Connection
is set high if no connection can open to the server.
Possible causes:
- There are too much clients connected on the server.
- All SAP are used for other protocols like MPI or other.

Bit 28: Range Error
Set high if the SRXM or STXM instructions indicate an element address (source or destination address) lying outside the permitted range.
Cause : Error in user program
Ranges monitored :

Inputs/outputs	0..8191
Flags	0..8191
Timers/counters	0..1599
Registers	0..8191

Example : During execution of the following STXM instruction, the Range Error bit is set high.

```
    STXM  1      ;channel 1
          25     ;25 registers
          R 1000 ;base address source
          R 8172 ;base address destination
```

An attempt is made to transmit the contents of registers 1000 to 1024 in the master station to registers 4072 to 4096 in the slave station.

Bit 29:TxNode Error
Set high if the node doesn't exist in the node list. The given node is not configured.

Bit 31: Program Error
Set high during execution of an STXM or SRXM instruction if the interface has been assigned in SS1 mode, or if a similar instruction is already executing (TBSY flag was not polled before executing the instruction).

[<sap_def>]
Master option to define the FDL sap number

Format: `"SAP:<Sap>;"`

	Value	Description
<Sap>	1..39	FDL sap number. Must be different than Slave SAP (Default SAP 12)

[<tout_def>]
Master option to define the Timeout value.
The default timeout for the connection is 1 second and can be changed using a SYSWR instruction:

```
    SYSWR    6100
             Timeout    ;Timeout value (in seconds)
```

Format: `"TOUT:<timeout>;"`

	Value	Description
<timeout>		Timeout value.(Default 250ms)

Examples of SASI Texts

Profi-S-Bus (Master)

```
    $SASI
    TEXT 60   "MODE:PSM,R555;DIAG:F8000,R4005;"
    $ENDSASI
```

Profi-S-Bus (Slave)

```
    $SASI
    TEXT 60   "MODE:PSS;DIAG:F8000,R4005;"
    $ENDSASI
```

See also
Using symbols in Texts

9.15 SASI Text (Ether-S-Bus)

Description
The master and slave channel must be configured from device configurator and completed with a special text definition for the SASI instruction:

- The master SASI text defines the diagnostic flags, register address and options necessary to the master communication instructions: STXM/SRXM/...
- The slave SASI text is not necessary if the user don't need the diagnostic flags and registers. All definitions necessary are already present in the Device Configurator.

Format

```
                                   ;xxxx is a Text number
TEXT xxxx     "<MODE_DEF>;"        ;comms mode and Register for slave station number
              "<DIAG_DEF>;"        ;diagnostic elements for Ethernet communication
              "<OPTION_DEF>;"      ;options for Ethernet communication
```

Validating SASI texts
The $SASI and $ENDSASI directives can used to enclose SASI texts. This causes S-Asm to check the syntax of the text and all characters are converted to uppercase.

<MODE_DEF>

Defines communications mode and a register for the slave station number.

Format : `"MODE:<sbus_mode>[,<dest_reg>];"`

<sbus_mode>	**Description**
EM	Ether S-Bus master mode.
ES	Ether S-Bus slave mode (only used for slave diagnostics). Note: The Ethernet slave is configured automatically when configurating the TCP/IP module in the PG5 Device Configurator.

<dest_reg>	Master: this register defines the address of the partner station. (Type: Rxxxx) Slave: do not defined

The remote station address has to be stored in the register before to send the commands STXM/ SRXM.
This addressing uses two address fields, the upper and the lower part of the address register.

Register address field (32 bit)		
More significant word	Less significant word	
IP node number	Not used	S-Bus address

Example:

```
LDL     R 500      ; partner address register
        12         ; S-Bus address 12
LDH     R 500
        15         ; IP node number 15
STXM    9          ; Interface no. 9
        16         ; Transmit 16 elements
        R 150      ; Register 150..165
        C 500      ; to counters 500..515
```

<DIAG_DEF>

Defines the diagnostic elements for the Ether-S-Bus communication.

Format: `"DIAG:<diag_elem>,<diag_reg>;"`

	Type	**Description**
<diag_elem>	F xxxx O xxxx	Base address of 8 consecutive Flags (or Outputs)
<diag_reg>	R xxxx	Address of a register for diagnostic

The 8 Flags give information about the status of the channel. In case of error when executing an communication instruction, more information can be obtained by examining the contents of the Diagnostic Register.

Diagnostic Flags

The SASI texts DIAG address is the base address of 8 consecutive Outputs or Flags, used as follows:

Address	Name	Description
xxxx	RBSY	Receiver busy
xxxx+1	RFUL	Receive buffer full
xxxx+2	RDIA	Receiver diagnostic
xxxx+3	TBSY	Transmitter busy
xxxx+4	TFUL	Transmitter full
xxxx+5	TDIA	Transmitter diagnostic
xxxx+6	XBSY	SASI permission
xxxx+7	NEXE	Not executed

Receiver Busy (RBSY)
Set high when a slave station receives a telegram. The flag is reset as soon as the reply telegram has been sent. This flag has no significance in the case of the master station

Receive Buffer Full (RFUL)
Set high when elements in the slave station have been changed by the master station.

Receiver Diagnostic (RDIA)
Set high when an error is noticed during receipt of a telegram. A detailed description of the error can be obtained from the diagnostic register (bits 0..15).
After execution of a communication instruction, RDIA is reset only if all receiver diagnostic bits (0...15) in the diagnostic register are 0.

Transmitter Busy (TBSY)
Set high while transmission is taking place.
Master station :
It is set high during execution of an STXM or SRXM instruction. The flag is reset as soon as a valid reply is received.
Slave station :
It is set high while the reply is transmitted.

Transmitter Diagnostic (TDIA)
(TDIA) is set high if an error occurs during transmission of a telegram. A detailed description of the error can be obtained from the diagnostic register. After execution of a communication instruction, TDIA is reset only if all transmitter diagnostic bits (16...31) in the diagnostic register are 0.

Interface busy (XBSY)
Low when the user has the permission to perform a SASI OFF to undo the S-Bus PGU for Public Line modem.

Not Executed (NEXE)
Set high if an instruction (STXM or SRXM) has not been completed after three attempts. The flag is

reset by the next S-Bus instruction.

Diagnostic Register

If the diagnostic flag TDIA or RDIA is high, the diagnostic register content will help you to found the communication trouble.
Any bit which has been set high in the diagnostic register remains so, until manually reset by the user program or the debugger.

	Bit	**Designation**	**Description**
	0	Overrun error	Overrun of the internal receiver buffer
	1		
	2	Framing error	Usually caused by an incorrect baud rate
	3	Break error	Break in data line
R	4	BCC error	Bad Block Check Code or CRC-16
E	5		
C	6		
E	7		
I	8	Length error	The telegram length is invalid
V	9		
E	10	Address error	Address of ACK is invalid
R	11		
	12	Range error	Invalid element address
	13	Value error	Error in the received value
	14	RxBroadcast_error	Error while receiving a not alowed broadcast telegram over IP.
	15		
	16	Retry count	Indicates the number of retries (in binary)
	17		(telegram repeats in binary representation)
T	18		
R	19		
A	20	NAK response	Negative response (NAK) was received
N	21	Missing response	No response was received after timeout
S	22	Multiple NAK	NAK received after retries
M	23	Target not present	Target station not present
I	24		
T	25		
T	26		
E	27		
R	28	Range error	Invalid element address
	29	TxNode_error	Node does not exist
	30	TxBroadcast_error	Error when sending broadcast tlg over IP
	31	Program error	Attempt to transmit when unauthorised

Bit 0: Overrun Error
Set high when there is an overrun of the internal buffer of the DUART.
Cause : Baud rate assigned is too high ? the CPU can no longer process all characters received. This can happen if one CPU is involved in communications requiring a high rate of data transmission via several interfaces simultaneously. It is theoretically possible for all interfaces of a CPU (excluding the 20mA current loop) to be assigned the maximum Baud rate of 19.200 bps at the same time. In practice, however, this error can arise when there is a very high level of communication over several interfaces. The system program handles the interfaces with differing priorities. The highest priority is allocated to interface 0, declining to interface 3.
Remedy :
- Reduce Baud rate.
- For fast communication, use an interface with high priority, if possible.

Bit 2: Framing Error
Set high when a character is received with a framing error (missing stopbit). This is usually caused by setting the Baud rate wrongly.

Bit 3: Break Error
Set high when an interruption is noticed during receipt of a character.
Cause : Data line broken or wrongly set Baud rate.

Bit 4: BCC or CRC-16 Error
Set high if a CRC-16 error is identified on the incoming telegram. The incoming telegram is rejected.
Reaction of Slave: the received telegram will be ignored
Reaction Master: the received telegram will be ignored and the last telegram will be retransmitted.
Cause : Interference on the data line.
Remedy : Check electrical installation.

Bit 8: Length Error
Set high when a telegram is received with invalid length. This error cannot arise in a network made up exclusively of PCD stations. The error indicates that an invalid telegram has been received from an external system. This results in a NAK response.

Bit 10: Address Error
Set high if an invalid telegram is received (incorrect command code).
Cause : Same as for Length Error (there is no NAK response).

Bit 11: Status Error
Set high when the PCD can not execute a command request because the slave PCD is not in the correct status (Run/ Halt/Stop/Diconnected/…). Only used for S-Bus level 2

Bit 12: Range Error
Set high if an incoming telegram contains an invalid PCD element address. This error cannot arise in a network made up exclusively of PCD stations, as the master PCD monitors the element address range of telegrams as they are transmitted. The slave station responds to this error with NAK.

Bit 13: Value Error
Set high when an invalid data value is received.
Example :
The STXM instruction is used in an attempt to load the clock. The value received for the hour is 30. However, the maximum range for the hour is only 0..23.
The slave station responds to this error with NAK.

Bit 14: Missing Media Error
Set when an invalid broadcast telegram is received (IP broadcast ? IP node = 65535 and S-Bus address < 255).

Bit 16 and 19: Retry Count (Bits 16 and 19)
Shows the number of repeat telegrams sent during execution of an SRXM or STXM instruction, represented in binary. Bit 16 is the LS bit. The quality of an S-Bus network can be judged by monitoring these two bits.

Bit 20: Negative Response
Set high if a NAK response is received from a slave. This means that the master has previously sent an invalid telegram. Check for the following errors: Value Error, Range Error and Length Error.

Bit 21: Missing Response
Set high if no response has been received from the slave station after the time-out has elapsed.
In this case, the telegram is retransmitted (maximum two times).
Possible causes :
- The slave station addressed does not exist.
- Installation error in network (wiring).
- The slave station has received a confused telegram with a CRC-16 error.

Remedies :
- Check slave station (connections, station number)
- Have the correct line termination and pull-up/down resistors been connected on the bus line at the first and last stations ?

Bit 22: Multiple NAK
Set high if, instead of the expected ACK or NAK, a different response is received from a slave station.
Possible causes :
- More than one slave with the same station number.
- More than one master in the network.
- Interference on the bus line.

Remedies :
- As for Missing Response error

Bit 23: CTS Timeout
Set high if the target station can not be reached in the network Connection cable defect or power interrupted to the station.

Bit 28: Range Error
Set high if the SRXM or STXM instructions indicate an element address (source or destination address) lying outside the permitted range.
Cause : Error in user program
Ranges monitored :

Inputs/Outputs	0..8191
Flags	0..8191
Timers/Counters	0..1599
Registers	0..8191

Example : During execution of the following STXM instruction, the Range Error bit is set high.
```
    STXM    1        ;channel 1
            25       ;25 registers
            R 1000   ;base address source
            R 8172   ;base address destination
```
An attempt is made to transmit the contents of registers 1000 to 1024 in the master station to registers 4072 to 4096 in the slave station.

Bit 29: TxNode Error
Set high if the node does not exist in the node list, or if it has not been configured, or if an invalid broadcast telegram has been sent (IP broadcast ® IP node = 65535 and S-Bus address < 255).

Bit 30: TxBroadcast Error
Set high if an invalid broadcast telegram has been sent (IP broadcast ® IP node = 65535 and S-Bus address < 255).

Bit 31: Program Error
Set high during execution of an STXM or SRXM instruction if the interface has been assigned in SS1 mode, or if a similar instruction is already executing (TBSY flag was not polled before executing the instruction).

<OPTION_DEF>

Defines the option elements for the Ethernet communication.

Format: `"[<tout_def>],[<dbx_def>];"`

	Type	Description
`< tout_def>`	`TOUT:xxx`	Is the timeout value of the EM mode in ms. Default timeout is 500ms (lowest limit value 100 ms).

`< port_def>`	`PORT:xxx`	Optional selection of S-Bus communication port:
		Default = 0
		1 = automatic allocated port 1024 … 4999
		X = 5000 … 65'535

Examples of SASI Texts

Mode Ether-S-Bus (Master)

```
$SASI
TEXT 100  "MODE:EM,R100;DIAG:F1000,R1000;TOUT:500"
$ENDSASI
```

Mode Ether-S-Bus (Slave)

```
$SASI
TEXT 101  "MODE:ES;DIAG:F2000,R2000"
$ENDSASI
```

See also
Using symbols in Texts

9.16 SASI Text (Profibus-DP)

Description
The SASI text is generated by the PROFIBUS-DP configurator and has the following format:

Master:
`"MODE:DPM;CONF:DBXxxxx;DIAG:Fyyyy,Rzzzz"`

Slave:
`"MODE:DPS;CONF:DBXxxxx;DIAG:Fyyyy,Rzzzz"`

xxxx	Specific number of a DBX containing all PROFIBUS-DP information.
yyyy	Specific number of the first diagnostic flag or diagnostic output
zzzz	Specific number of the first diagnostic register.

Diagnostics

Diagnostics of a PROFIBUS-DP communication takes place in the usual way for the PCD, i.e. for each communications channel, 8 flags are assigned for rough diagnosis and up to a maximum of 70 registers for fine diagnosis.
The diagnostic data addresses are defined in the Device Configurator.

Diagnostic flags with PROFIBUS-DP

Address	Name	Description
xxxx	SLAVE_ERR	Slave error Error in the slave
xxxx+1	GCS_BUSY	Global Control Service is processing
xxxx+2	SERV_BUSY	Service function is processing
xxxx+3	DATA_EXCH	Data exchange Exchange of data between master and slave
xxxx+4		Not used
xxxx+5		Not used
xxxx+6	CONF_RCV	Configuration received Slave has received a configuration telegram from the master
xxxx+7	CONF_STAT	Configuration status Indicates whether configuration data is OK

Slave_error (SLAVE_ERR)

Master:	H = Error in one or more slaves
	L = No error in slaves
Slave:	H = Error in slave
	L = No error in Slave

Master:
The number of the slave that generated the error can be obtained from diagnostic registers +3 to +6.
This flag is set low when, after completion of a 'Read slave diagnostic data' telegram, there are no longer any errors present.

Global Control Service (GCS_BUSY)

Master:	H = Global Control Service is busy
	Global Control Service has finished
Slave:	Not used

Global Control Services are: Freeze, Unfreeze, Sync and Unsync.

Service (SERV_BUSY)

Master:	H = Service function is busy L = Service function has finished
Slave:	Not used.

Service functions are:
- Stop data exchange between the PCD controller's process image memory and PROFIBUS-DP card memory.
- Read slave diagnostic data.
- Activate or deactivate a slave.

Data Exchange (DATA_EXCH)

Master:	H = Data exchange on the PROFIBUS-DP network is running. L = Data exchange on the PROFIBUS-DP network has halted.
Slave:	H = Connection with master established (executing data exchange). L = No data exchange connection with master. The flag becomes = L only after the watchdog time is elapsed

Configuration received (CONF_RCV)

Master:	Not used.
Slave:	H = Slave has received a configuration telegram from master. L = Slave has not received a configuration telegram from master.

Configuration status (CONF_STAT)

Master:	Not used.
Slave:	H = The configuration telegram from the master corresponds to the slave configuration. L = The configuration telegram from the master does not correspond to the slave configuration.

Diagnostic Registers with PROFIBUS-DP
Diagnostic registers are grouped by the following areas:

- Service area
- Station area
- Standard PROFIBUS-DP diagnostic area
- Expanded PROFIBUS-DP diagnostic area

The maximum size of diagnostic registers is defined by the 'Max_Diag_Data_Len' parameter from the slave device GSD file, since slave diagnostic data is stored in the diagnostic registers.

'Max_Diag_Data_Len' can have a maximum size of 244 bytes. When there is more than one slave, the largest 'Max_Diag_Data_Len' parameter always applies.

At present, the diagnostic registers are only used by the master.
Division of diagnostic registers:

Areas	Address		Description
Service area	Base +0		Result of Global Control Service GCS
	Base +1		Result of IL instruction SCON(I) Fct. 0,1,8,9

	Base +2		Result of IL instruction SCON(I) Function #7
	Base +3		Error status station 0…31
Station area	Base +4		Error status station 32…63
	Base +5		Error status station 64…95
	Base +6		Error status station 96…126
Standard	Base +7		Length of PROFIBUS-DP diagnostic (byte 6…243)
Profibus- DP	Base +8		Standard DP diagnostic (byte 0 and 1)
Diagnostic	Base +9		Standard DP diagnostic (byte 2 … 5)
	Base +10		Expanded DP diagnostic (byte 6…9)
Expanded	Base +11		Expanded DP diagnostic (byte 10…13)
Profibus DP	Base +12		Expanded DP diagnostic (byte 14…17)
Diagnostic	Base +13		Expanded DP diagnostic (byte 18...21)
/ /		//	
	Base +69		Expanded DP diagnostic (byte 242 and 243)

Description of Diagnostic Registers

Result GCS (base + 0)
In this register the result of the 'Global Control Service' is stored. The 'Global Control Service' is triggered by function codes 13..16 of the SCON instruction.
The result codes are the same as described under: 'Result of IL instruction SCON(I) Fct. 0, 1, 8, 9 (Base + 1)'.

Result of IL instruction SCON(I)
Fct. 0, 1, 8, 9 (base + 1)
In this register the results of the following functions are stored:

- Run / Stop Data Exchange
  SCON wit function code 0.
- Read slave diagnostics.
  SCON with function code 1.
- Activate or deactivate slave.
  SCON with function code 8 or 9.

The following values are possible here:

Word	**Description**
0	Instruction has been successfully completed
1	Incorrect parameter (contact your local Saia Burgess Controls agent)
2	Not possible (contact your local Saia Burgess Controls agent)
3	No local resources (contact your local Saia Burgess Controls agent)
4	DP error (contact your local Saia Burgess Controls agent)
5	Slave is not OK
6	Not defined
7	Status conflict (contact your local Saia Burgess Controls agent)
8	Error in acyclic master-slave data exchange (contact your local Saia Burgess Controls agent)

20	Timeout
21	Station number does not exist
22	Instruction executed more than once (Diag Flag base+2 has not been checked)
23	Incorrect DP response
24	Incorrect parameter

Result of IL instruction SCON(I) Fct. 7 (base + 2)
The result of the following functions are stored in this register:

- Read station status.
  SCON with function code 7.

The register is coded here as follows:

31..8	Not used
7	System diagnostic flag (error)
6	Data exchange flag
5..4	Reserved
3..0	4-bit status, decimal value has these meanings: • 0: Cyclical data exchange running • 1: Error in connection • 2: Connection broken • 3: Stopped • 4: Slave deactivated • 5: Slave not defined • 6..15: Not used

Error status stations 0…31 (base + 3)
Each bit in this register corresponds to the station number of a slave device. As soon as an error occurs in a slave device, the relevant bit is set high.
The bit is set low when, after completion of a 'Read slave diagnostic data' telegram, there is no longer any error present.

31	Slave 31
30	Slave 30
…	
1	Slave 1
0	Slave 0

Error status stations 32…63 (base + 4)
Same function as for diagnostic register (base + 3) with errors for stations 32 to 63.

Error status stations 64…95 (base + 5)
Same function as for diagnostic register (base + 3) with errors for stations 64 to 95.

Error status stations 96…125 (base + 6)
Same function as for diagnostic register (base + 3) with errors for stations 96 to 125.

Length of PROFIBUS-DP diagnostic bytes 6…243 (base +7)
In this register, after an SCON instruction with function 1, the total length of diagnostic data (standard PROFIBUS-DP + external PROFIBUS-DP diagnostic) is stored in bytes. The length of diagnostic data differs in each slave device, amounting to no less than 6 bytes and no more than 244 bytes.

Standard DP diagnostic: bytes 0 and 1 (base +8)
In this register the first two bytes of standard PROFIBUS-DP diagnostic data are stored. Division into diagnostic registers is as follows:

Register# = Base +8			
3	2	1	0
Not used	Not used	DP byte 0	DP byte1

DP diagnostic byte 0, register bits 15..8:

15	Diag.deactivated: (set master)
14	Reservec
13	Sync_mode: Sync command received
12	Diag.freeze_mode: Freeze command received
11	Diag.WD_ON: Response monitoring active
10	Always 1
9	Diag.Stat_diag: Static diagnosis (Byte Diag-Bits)
8	Diag.Prm_req: Slave parameters must be reset

DP diagnostic byte 1, bits 7..0:

7	Diag.master_lock: (set Master) Slave parameters set by another master
6	Diag.prm_fault: Incorrect parameter set (Ident number etc.)
5	Diag.invalid_slave_response: (set slave fixed to 0)
4	Diag.not_supported: Requested Fct. is not supported in slave.
3	Diag.ext_diag: Slave has external diagnostic data.
2	Diag.cfg_Fault: Configuration data does not match.
1	Diag.station_not_ready: Slave is not ready for data exchange.
0	Diag.station does not exist(set master)

Standard DP diagnostic: bytes 2 to 5 (base +9)
In this register bytes 2 to 5 of the standard PROFIBUS-DP diagnostic data are stored.
The division is as follows:

Register# = Base +9			
3	2	1	0
DP byte 2	DP byte 3	DP byte 4	DP byte 9

DP diagnostic byte 2, bits 31..24:

31	Diag.ext_overfl
30..24	Reserved

DP diagnostic byte 3, bits 23..16:

23..16	Diag.master_add: Master address after parameter setting (FF without parameter setting)

DP diagnostic byte 4, bits 15..8:

15..8	Slave ident number, high byte

DP diagnostic byte 5, bits 7..0:

7..0	Slave ident number, low byte

Expanded DP diagnostic: bytes 6 to 9 (base +10)
In this register bytes 6 to 9 of the expanded PROFIBUS-DP diagnostic are stored.
The division is as follows:

Register# = Base +10			
3	2	1	0
DP byte 6	DP byte 7	DP byte 8	DP byte 9

DP diagnostic byte 6, bits 31..24:

31..24	Length of expanded diagnostic in bytes

DP diagnostic bytes 7, 8 and 9, bits 23..16, 15..8, 7..0

23..0	Meaning of bits must be obtained from slave descriptions.

Expanded DP diagnostic: bytes X0 to X3 (base +Z)
In these registers the expanded diagnostic information is stored.
The division is always as follows:

Register# = Base +Z			
3	2	1	0
X0	X1	X2	X3

9.17 $SASI..$ENDSASI

Description
These assembler directives can be used to delimit texts which are used by the SASI instruction.
All texts enclosed within these directives are checked by the assembler and any errors detected.
If $SASI .. $ENDSASI are not used, it is possible to enter an invalid text which may cause incorrect initialization of the channel (Error flag set).

Format
```
$SASI
<SASI Text definition>
$ENDSASI
```

Example
```
XOB     16
...
SASI    0       ;initialize serial channel 0
        100     ;using text 100
...
EXOB

$SASI
; Text 100 is checked as SASI text by the Assembler
TEXT 100  "UART:9600,7,E,1;MODE:MC0;DIAG:F1000,R4000;"
$ENDSASI
```

Note
If $SASI .. $ENDSASI are not used, it is possible to enter an invalid text which may cause incorrect initilialisation of the serial channel.

New SASI with '$'
SASI text accepts $
e.g.: `"UART:$Ra,$Rb,$Rc,$Rd;MODE:$Re,$Rf;DIAG:F$Rg,R$Rh;"`

Ra	Baudrate	110 .. 38400 (numeric)
Rb	Bits	7, 8 (numeric)

`Rc`	Parity	E, O, N (ASCII coded)
`Rd`	Stop	1 or 2 (numeric)
`Re`	Mode	'MC0', 'SM2' etc. (ASCII coded)
`Rf`	Station	Register with S-Bus station (numeric)
`Rg`	Diagnostic Flags	Register with the base Diagnostic Flag number (0 .. 8191 numeric)
`Rh`	Diagnostic Register	Register with the Diagnostic Register number (0 .. 4095 numeric)

9.18 Using Symbols in $SASI Texts

Description
Symbols can also be used in SASI texts. The value and optionally the type of the symbol is inserted into the text.
The symbol is written outside the text which is in double quotes, and must be separated from the text and other symbols by a comma.
After the symbol, an optional field width and prefix type can be given.

Format

`symbol [. [ [] [0] width] [t | T] ]`	
`symbol`	The symbol name. This can actually be any expression which includes a symbol, for example: `MotorOn + 100`, ... Symbols with floating point values are not permitted.
`.`	The dot immediately after the symbol indicates that a field width and/or a prefix is present.
`width`	The field width: the number of digits or spaces required for the number. If the width begins with a 0, leading zeros are inserted.
`t | T`	Optional prefix type `t` or `T`. If `t`, the value is prefixed with the symbol's type in lower case (o, f, r, ...). If `T`, the symbol's type is in upper case (O, F, R,...)

Example

```
BAUD      EQU  9600
D_FLAGS   EQU  F 500
D_REG     EQU  R 4095

XOB       16
SASI      1
          3999

TEXT 3999 "UART:", BAUD, ",7,E,1;MODE:MC0;"
          "DIAG:", D_FLAGS.T, ",", D_REG.T, ";"

EXOB
```

The resulting text will be:

```
TEXT 3999   "UART:9600,7,E,1;MODE:MC0;"
            "DIAG:F500,R4095;"
```

See also
SASI Instruction

9.19 SASI Mode OFF

Description
If the channel has been configured by a SASI instruction, then a new SASI instruction will fail unless a SASI "MODE:OFF" command is done.

Format
```
TEXT xxxx  "MODE:OFF"
```

See also
SASI Mode OFF on S-Bus PGU Slave

9.20 SASI Mode OFF on S-Bus PGU Slave

Description
If the channel has been configured by the Device Configurator as a PGU channel , then the SASI instruction will fail unless a SASI "MODE:OFF" command is done.

Format
```
TEXT xxxx "MODE:OFF,xx,yy,zz"
```

xx	timeout before the channel is unassigned: [0;300] [S]
yy	another SASI instruction must be executed before this timeout is over: [0;5000] [MS]
zz	not used by FW: [0;1]

See also
SASI Mode OFF

Practical example

```
XOB     16
SASI    1
        DIAG       ;"DIAG:F0,R0" assign diag flags and register
...
EXOB

COB     0
        0
...
SASI    1
        OFF        ;"MODE:OFF,..."
STH     XBSY
JR      H -1
SASI    1
        UserSASI   ;assign new mode
...
ECOB
```

- The XBSY flag is set to 1 after executing the SASI MODE:OFF, while the timeout xx has not elapsed.
- A new SASI (except SASI DIAG) can only be executed after a SASI MODE:OFF.
- If the yy timeout elapses before a new SASI is executed, the channel will reassigned as SBus PGU

slave.
- A SASI without the “MODE:OFF” text will give an error on SBus PGU slave.

9.21 SRXD - Receive Character (Mode C)

Description
Loads the next character (byte) present in the Receive Buffer of the channel given by the 1st operand into the Register given by the 2nd operand.
The instruction SRXD should be executed only if there is a character ready, indicated by RBSY = H otherwise the Error flag is set.
After SRXD is executed, the least significant 8 bits of the Register contain the character, all other Register bits are set to 0.
Up to 512 characters can be in the Receive Buffer. Each time SRXD is executed, the next character is read.
If the Receive Buffer overruns (more than 512 characters), then there will be a receive error (the RDIA flag and the corresponding status bit in the channels Diagnostic Register are set).

Format
```
SRXD[X]    [=] channel    ;channel number
           [=] reg (i)    ;Register to receive the character
```

Example
```
SRXD     3       ;read a character from channel 3
         R 100   ;and store it in Register 100
```

Flags

ACCU		Unchanged
Status Flags	E	Error flag set if the SRXD instruction is executed with an empty receive buffer or if the channel has not been correctly initialized or does not exist.

See also
SASI Assign Serial Interface
STXD Transmit Character (Mode C)
STXT Transmit Text (Modes C)

Practical example
Typical application in a Bloctec structured program:

```
    ...
    STH     F RBSY         ;if there is a character waiting
    CFB     H READ_CHAR    ;then read this character
    ...

    FB      READ_CHAR      ;FB to read a character
   [STH     H RDIA]        ;if there is a Receive Error
   [CFB     H RCV_ERROR]   ;then handle the error
    SRXD    0              ;read the character on channel 0
            R 999          ;and store it in R 999
    ...
    EFB
```

Note
In simple non-critical applications the error handling (above between brackets) can be omitted.

9.22 STXD - Transmit Character (Mode C)

Description

The character held in the least significant 8 bits of the Register given in the 2nd operand is placed in the Transmit Buffer of the serial channel given by the 1st operand. It is then transmitted automatically.

The Transmit Buffer can hold up to 512 characters. If it is empty (all characters have been transmitted), the TBSY status flag is set Low. While there are characters waiting to be transmitted, TBSY remains High.

If the TDIA status is High after executing an STXD, this indicates a problem, and the Diagnostic Register should be examined.

Format

```
STXD[X]   [=] channel     ;channel number
          [=] reg  (i)    ;Register containing the character to transmit
```

Example

```
STXD    1        ;transmit the character in
        R 100    ;Register 100 (bits 7..0) on channel 1
```

Flags

ACCU		Unchanged
Status Flags	E	Error flag set if the channel has not been correctly initialized or does not exist.

See also

SASI Assign Serial Interface
SRXD Receive Character (Mode C)
STXT Transmit Text (Modes C)

Practical example

Typical application in Bloctec structured program:

```
    ...
    STL     F TFUL         ;if there is room in the TX buffer
    CFB     H SEND_CHAR    ;then send a character
    ...

    FB      SEND_CHAR      ;FB to send a character
    STXD    0              ;send on channel 0
            R 900          ;the character stored in R 900
   [STH     H TDIA]        ;if there is a Transmit Error
   [CFB     SND_ERROR]     ;then handle the error
    ...
    EFB
```

Note

In simple or non-critical applications the error processing (above between brackets) can be omitted.

9.23 STXT - Transmit Text (Mode C)

Description

Transmits the Text indicated in the 2nd operand via the serial channel given by the 1st operand. Status bit XBSY is set High, and the PCD transmits the Text. XBSY is set Low when the Text has been transmitted.

The normal execution of the program is not affected because the Text is transmitted as a background operation.

Texts can contain control strings to allow the formatted transmission of data values, see Texts Containing Data and Text Output Formats.

The XBSY flag indicates the completion of the background task. Whilst XBSY is High no other communications instruction should be performed on this serial channel.
The NEXE diagnostic flag is set if the Text contains a bad control string.

Format
```
STXT[X]   [=] channel    ;channel number
          [=] text  (i)  ;text number to transmit
```

Example
```
STXT     0    ;transmit Text 123 on serial channel 0
         123
```

Flags

ACCU		Unchanged
Status Flags	E	Error flag set if the channel has not been correctly initialized or does not exist.

See also
SASI Assign Serial Interface
SRXD Receive Character (Mode C)
STXD Transmit Character (Mode C)
Texts Containg Data (Mode C)
Text Output Formats (Mode C)
Using Symbols in Texts

Practical example
When Input 1 goes High, the following text should be sent: "Abort!".

```
    XOB     16
    SASI    1        ;initialize serial channel 1
            0        ;with parameters stored in Text 0
    EXOB

    $SASI
    ;9600 Baud, 7 Data bits, Even Parity, 1 Stop bit
    ;Mode MC0, Diagnostic flags: F1000..F1007, Diagnostic register: R1000
    TEXT 0  "UART:9600,7,E,1;MODE:MC0;DIAG:F1000,R1000;"
    $ENDSASI

    TEXT 10 "Abort!<CR><LF>"

    COB     0
            0
    STH     I 1      ;if Input 1 goes High
    DYN     F 0      ;(rising edge detection)
    ANL     F 1006   ;and not already transmitting (F1006 = XBSY)
    JR      L End
    STXT    1        ;then send Text 10 over serial channel 1
            10
End:
    ECOB
```

9.24 Texts Containing Data (Mode C)

Transmitted texts can contain data such as the clock value, the state of an Input, the contents of a Register, etc. This is done by using a special character sequence in the Text, beginning with $ or @, as shown in the table below.
Values can also be formatted for field width, left/right justified etc, see Text Output Formats (Mode C).

NOTE
In Mode C texts, media addresses must always be 4 digits, or with firmware version 1.20.0 or later 5 digits can be used for addresses > 9999.
To ensure 4 or 5 digits, use the format `Symbol.04T`, see Using symbols in Texts.
    `"$", Symbol.04T` or `"$R", Symbol.04`

If the symbol has a 5-digit address, S-Asm will automatically use 5 digits and insert an `X` character so the firmware knows it's 5 digits. This means you can still use the `Symbol.04T` format for 5-digit addresses.

For example:

```
;This creates text "Register 1234 = $R1234<CR><LF>"
Symbol1 EQU R 1234    ;4 digit address
TEXT 4000 "Register ", Symbol1, " = $", Symbol1.04T, "<CR><LF>"

;This creates text "Register 12345 = $RX12345<CR><LF>"
Symbol2 EQU R 12345   ;5 digit address
TEXT 4001 "Register ", Symbol2, " = $", Symbol2.04T, "<CR><LF>"

;This also creates text "Register 12345 = $RX12345<CR><LF>". but
;the media type must be in the text, the format does not have 'T':
Symbol3 EQU R 12345   ;5 digit address
TEXT 4003 "Register ", Symbol2, " = $RX", Symbol2.05, "<CR><LF>"
```

$ = Direct Addressing

Absolute media address is provided.

$H	Time (Hour,Minute,Second): hh:mm:ss	
$HH	Time (Hour only): hh	
$HM	Time (Minute only): mm	
$HS	Time (Second only): ss	
$D	Date (Year, Month, Day): yy-mm-dd	
$d	Date (Day, Month, Year): dd.mm.yy	
$DD	Date (Day only): dd	
$DM	Date (Month only): mm	
$DY	Date (Year only): yy	
$W	Week (Week number, Day of week): ww-dd	
$WN	Week (Week number only): ww	
$WD	Week Day (Day number only): dd	
$innnn	Logical state of a single Input (0, 1)	nnnn = media address
$onnnn	Logical state of a single Output (0, 1)	(must be 4 digits)

`$fnnnn`	Logical state of a single Flag (0, 1)	
`$Innnn`	Logical state of 8 Inputs (nnnn to nnnn+7)	`nnnn` = first media address
`$Onnnn`	Logical state of 8 Outputs (nnnn to nnnn+7)	(must be 4 digits)
`$Fnnnn`	Logical state of 8 Flags (nnnn to nnnn+7)	
`$Cnnnn`	Counter contents	`nnnn` = media address
`$Rnnnn`	Register contents	(must be 4 digits)
`$Tnnnn`	Timer contents	
`$Lnnnn`	Includes another Text (max. 3 levels) See also the new `$Innnn` below	`nnnn` = Text number (must be 4 digits)
`$xnn`	Character '`x`' is repeated '`nn`' times The character cannot be one of these: `$ : H D d W i o f I O F C R T L A`	`nn` must be 2 digits
`$Annnn`	Output Register contents as ASCII character	`nnnn` = Register number (must be 4 digits)

Example of $Annnn:

`"$A0999"`	when R 999 = 00000000 hex	`'NUL'`
`"$A0999"`	when R 999 = 00000061 hex	`'a'`
`"$A0999"`	when R 999 = 00006162 hex	`'ab'`
`"$A0999"`	when R 999 = 00616263 hex	`'abc'`
`"$A0999"`	when R 999 = 61626364 hex	`'abcd'`

Preceding zeros are not output. An ASCII zero is only output if the lowest value byte is equal to 0.

New formats for firmware version 1.20.00 and later

`$bxxxx.yyyyy`	Data Block element	`xxxx` = DB number `yyyyy` = element number 0..16383, must be 5 digits
`$lnnnn` `@lnnnn`	Includes another Text but only up to the first <0> character in the Text. If the Text does not contain <0>, the entire Text is included.	`nnnn` = Text number, 4 digits

`$RXnnnnn`
`@RXnnnnn`
`$FXnnnnn`
	If an X character precedes the address, then a 5-digit address is assumed. Use this for Register and Flag addresses > 9999. Note: In some cases S-Asm will automatically insert the `X` if it knows the address is > 9999.	`X` = indicates 5 digits
		`nnnnn` = 5-digit R or F number

Examples of formats containing 5-digit R or F addresses > 9999, use an `X` after the data type:
`$RX16383`
`$FX16383`

@ = Indirect Addressing

The media address is supplied in a Register.

`@innnn`	Logical state of a single Input (0, 1)	`nnnn` = media address
`@onnnn`	Logical state of a single Output (0, 1)	(must be 4 digits)
`@fnnnn`	Logical state of a single Flag (0, 1)	
`@Innnn`	Logical state of 8 Inputs (add to add+7)	
`@Onnnn`	Logical state of 8 Outputs (add to add+7)	
`@Fnnnn`	Logical state of 8 Flags (add to add+7) Use `@FXnnnnn` for 5-digit address	
`@Cnnnn`	Counter contents	
`@Rnnnn`	Register contents Use `@RXnnnnn` for 5-digit address	
`@Lnnnn`	Includes another Text (max. 3 levels) See also `@lnnnn` above	
`@xnnnn`	Character '`x`' is repeated Register contents times. The character cannot be one of these: `@ : i o f I O F C R L`	

**NOTE: To output a single '$' use "$$", to output a single '@' use "@@".
Register numbers above 9999 cannot be used (more than 4 digits).**

Example 1

```
InpAdds EQU I 0
RegAdds EQU R 100

TEXT 10    "Date: $d  Time: $H<CR><LF>"
           "Input 0..7:    $", InpAdds.04T, "<CR><LF>"
           "Register 100:  $R", RegAdds.04, "<CR><LF>"
           "$+32<CR><LF>"
```

Assembling and linking produces this text:

```
TEXT 10    "Date: $d  Time: $H<13><10>"
           "Input 0..7:    $I0000<13><10>"
           "Register 100:  $R0100<13><10>"
           "$+32<13><10>"
```

Assuming that this text is printed on the 16th August 2012 at 09:32 am, that Inputs 0 and 1 are High, and the contents of Register 100 is 12345, the following will be printed:

```
Date: 16.08.12  Time: 09:32:59
Input 0..7:     11000000
Register 100:   12345
++++++++++++++++++++++++++++++++
```

Example 2

Practical use of "$A...."

Cursor position on a screen should be determined from 2 registers for the X and Y position:

- X position from Register R 1 (1..80)
- Y position from Register R 2 (1..25)

The escape sequence for cursor positioning is: <27><17><value for X><value for Y>
It is possible to program: `"<27><17>$A0001$A0002"`
Where the <value for X> is in Register 0001 and the <value for Y> is in Register 0002.

To output a fixed position of X = 40 and Y = 12, the whole sequence of 4 characters can be written into a single Register and output using $A....
Note that all values must be in hex format:
Esc = 1B hex; 17 = 11 hex; 40 = 28 hex (X value); 12 = 0C hex (Y value)

Load a Register with the data to send:

```
LD   R 1000
     1B11280Ch
```

Transmit this text to position the cursor::

```
"$A1000"
```

9.25 Text Output Formats (Mode C)

The format of transmitted Register and Counter data can also be specified in the Text.

The field width and number of decimal places can be specified. Format definitions are introduced by the text "$%xxxx", where 'xxxx' is the required format, see below.
If such a definition is output, all the following Register or Counter values are output using this format, until another format definition is encountered.

In the following format definitions, the d | D means 'decimal', x | X = hexadecimal and b | B = binary. Other number base formats are not supported. If the value is too large to fit in the defined field, default formatting is used (no formatting).

Output format definitions
Assume Registers 10, 11 and 12 contain respectively the following constant values: 123456, -7890 and 5.

No formatting (default)
The field width depends on the size of the number.

```
TEXT 0    "REGISTER 10: $R0010<10><13>"
          "REGISTER 11: $R0011<10><13>"
          "REGISTER 12: $R0012"
```

Output:

```
REGISTER 10: 123456
REGISTER 11: -7890
REGISTER 12: 5
```

Fixed width field
Use the format definition "$%xxd" or "$%xxD", where 'xx' (1..99) signifies the field width.
"$%xxd": The value is right justified with leading spaces.

```
TEXT 1    "$%08dREGISTER 10: $R0010<10><13>"
          "REGISTER 11: $R0011<10><13>"
          "REGISTER 12: $R0012"
```

Output:

```
REGISTER 10:   123456
REGISTER 11:    -7890
REGISTER 12:        5
```

"$%xxD": The value is right justified with leading zeroes.

```
TEXT 1    "$%08DREGISTER 10: $R0010<10><13>"
          "REGISTER 11: $R0011<10><13>"
          "REGISTER 12: $R0012"
```
Output
```
REGISTER 10: 00123456
REGISTER 11: -0007890
REGISTER 12: 00000005
```

Fixed width field and fixed number of decimal places

The value is right-justified, but the number of decimal places is always displayed, and is padded on the right with zeros.
Use the format definition "$%xx.yd", where 'xx' is the total field width, and 'y' is the number of places to the right of the decimal point.

```
TEXT 2    "$%07.3dREGISTER 10: $R0010<10><13>"
          "REGISTER 11: $R0011<10><13>"
          "REGISTER 12: $R0012"
```
Output:
```
REGISTER 10: 123.456
REGISTER 11:  -7.890
REGISTER 12:   0.005
```

Fixed decimal places only

The number of decimal places is fixed but the field width is dependent on the size of the number.
Use the format definition "$%00.yd", where 'y' is the number of decimal places, padded on the right with zeros if required.

```
TEXT 2    "$%00.5dREGISTER 10: $R0010<10><13>"
          "REGISTER 11: $R0011<10><13>"
          "REGISTER 12: $R0012"
```
Output:
```
REGISTER 10: 1.23456
REGISTER 11:-0.07890
REGISTER 12: 0.00005
```

Removing formatting

"$%00d" sets the standard format (no formatting).

Saving / Restoring format definitions

Format definitions may be saved using "$sn", where 'n' is a 'save' number.
Up to 10 format definitions can be saved (09).
Saved formats are restored using "$n", where 'n' is the 'save' number of the format definition to be restored.
Formats may be saved as part of the initialization process, in XOB 16, the startup XOB.
To save a format, the text containing this format must be output to the serial line with the STXT instruction.
If a format is restored which has not been saved, the default format (no formatting) is used.

Example:
```
    XOB     16                ;startup XOB
    ...

    TEXT 991 "$%05.1d$s1"     ;Format 1 definition (nnn.n)
    TEXT 992 "$%04.2d$s2"    ;Format 2 definition (n.nn)
    TEXT 993 "$%08.3d$s3"    ;Format 3 definition (nnnn.nnn)

    ;Activation of the format definitions
    SEI     K 0
Loop1:
    STH     XBSY
    JR      H DEF
    STXTX   0
            991
    INI     K 2
    JR      H Loop1
    ...

    EXOB

    COB     0
            0
    ...
    STXT    1
            10

    TEXT 10 "Pump Liters     Price/L    Total<10><13>"
            " 1   $1$R0010   $2$R0011   $3$R0012<10><13>"
            " 2   $1$R0013   $2$R0014   $3$R0015<10><13>"

    ...
    ECOB
```

Output:

```
Pump  Liters     Price/L    Total
1     13.8       0.86       11.868
2     158.2      0.95       150.290
```

Including other texts

The "$Lnnnn" sequence 'incLudes' another text which is processed as though it were part of the original text.
If this included text contains a new format definition, the format is used until the end of the text. On return to the original text, the original format definition is restored.

Example

```
    COB     0
            0
    ...
    STXT    1         ;send Text 10
            10
    TEXT 10 "$L0100 Motor speed too high<10><13>"
    ...
    STXT    1         ;send Text 11
            11

    TEXT 11 "$L0100 Oil pressure too low<10><13>"
    ...
    ECOB

    TEXT 100 "Diesel Engine ALARM:"
```

Result:
```
Diesel Engine ALARM: Motor speed too high
Diesel Engine ALARM: Oil pressure too low
```

9.26 SRXM - Receive Media (Mode S-Bus)

Description
This instruction reads data or status from a slave station and copies them into the master PCD.
The slave's station number must be loaded into the Register defined by the SASI Text, and the SASI instruction should be executed first to configure the channel.
This instruction can only be used in the master PCD.

The TBSY Flag is set High while it is being processed, and is reset once the data transfer is complete.
Before executing the SRXM instruction, the TBSY Flag can be tested to ensure it is Low.

The instruction uses of four lines:
The 1st operand is the channel number.
The 2nd operand defines the number of items to transfer.
The 3rd operand defines the base address (lowest) of the source data in the slave PCD.
The 4th operand defines the base address (lowest) of the destination data in the master PCD.

Format
```
SRXM[X]   [=] channel     ;channel number
          [=] count       ;number of items to receive
          [=] source (i)  ;base address of source data (in the slave)
          [=] dest   (i)  ;base address of destination data (in the master)
```

count	1..32	Number of R T C to read
	1..128	Number of I O F to read
	0	Special function code
	R nnnn	Used for Data Block transfer
source	Base address of data in the slave PCD	
	I O F	Input, Output, Flag
	R	Register
	T C	Timer, Counter
	DB	Data Block
	K	K 0..6000 special function code
dest	Base address of data in the master PCD	
	I O F	Input, Output, Flag
	R	Register
	T C	Timer, Counter
	DB	Data Block

Data source and destination
The following table shows which data can be copied from the slave station to the appropriate data in the master station.

	Master PCD (destination)						
		O	F	R	C	T	DB

	I	x	x				
	O	x	x				
Slave PCD	F	x	x				
(source)	R			x	x	x	x
	C			x	x	x	x
	T			x	x	x	x
	K			x			
	DB			x	x	x	

Special Function Codes

Code	**Function description**		**Examples of result**		
K 0..7	Read PCD status: 0..6: CPU number of slave PCD 7: Own CPU status		R C H S D	Run Conditional Run Halt Stop Disconnected	
K 1000	Read Clock		The contents of the clock is written in two Registers (same format as for RTIME instruction)		
K 2000	Read Display Register				
K 3000	Read Size of Data Block				
K 5000 K 5010	Read Device type	in ASCII in decimal	ASCII	Decima l	Type
			" D1" " D2" " D4" " D6"	1 2 4 6	PCD1 PCD2 PCD4 PCD6
K 5100 K 5110	Read Module type	in ASCII in decimal	ASCII	Decima l	Type
			" M1_" " M1_" " M15" " M11" " M12" " M14" " M24" " M34" " M44" " M1_" " M2_" " M3_" " M54"	10 10 15 11 12 14 24 34 44 10 20 30 54	PCD1.M1 PCD2.M12 PCD2.M15 PCD4.M11 PCD4.M12 PCD4.M14 PCD4.M24 PCD4.M34 PCD4.M44 PCD6.M1 PCD6.M2 PCD6.M3 PCD6.M5
K 5200 K 5210	Read Firmware version	in ASCII	Examples of valid responses : " $4C", " 004", " X41"		
		in decimal	E.g. : 5 dec for Version 005 -1 dec for any '$', 'X', 'β'		
K 5300	Read CPU number	in ASCII	ASCII	Decima	Type

K 5310		in decimal		I	
			"0" "0" "0" or "1" "0" to "6"	0 0 0 or 1 0 to 6	PCD1 PCD2 PCD4 PCD6
K 6000	Read S-Bus station number in BROADCAST This telegram is always transmitted in broadcast mode (address = 255). This will only work in point-to-point communication.				

Example

Read I/O/F/R/T/C in a slave station:

```
LD      R 100     ;load register defined in the SASI text
        10        ;with the slave station number 10
SRXM    1         ;read via channel 1
        20        ;20 items
        R 100     ;from R 100..119 of station 10
        R 0       ;into R 0..19
```

Special function to read the slave clock from the master station:

```
LD      R 100     ;load register defined in the SASI text
        12        ;with the slave station number 12
SRXM    1         ;read via channel 1
        0         ;(must be 0)
        K 1000    ;the slave's clock
        R 20      ;into master registers 20 and 21
```

Flags

ACCU	Unchanged
Status Flags E	Error flag set if channel not correctly initialized, or SRXM executed while already busy

Reading Data Blocks

The format of the SRXM instruction is slightly different for Data Blocks. To address a Data Block element, both the DB number and the element number are needed.

```
SRXM[X]   channel            ;channel number
          count_position     ;register containing number of elements and offset
          source      (i)    ;base address of source element (in slave) (note 1)
          destination (i)    ;base address of destination element (in master)
```

count+position	The number of a Register which contains the number of items to transfer (count) 1..32 in the MS word (bits 31..16), and the starting item number (position) in the LS word (bits 15..0). This Register can be loaded using LDL first to load the position and LDH to load the count.
source destination	The source and destination addresses must be compatible data types, see the table above.

Note 1) When using SRXMX in indexed mode, the source and destination are both indexed with standard media (I O F R T C),
but Data Blocks are not indexed.

Example

To transfer Registers 2000..2031 (32 items) from the slave station into Data Block 7999 position 1000 of the master station via channel 3.

```
LDL     R 100       ;initialize the Position in the DB
        1000
LDH     R 100       ;initialize the Count (number of Registers)
        32
SRXM    3           ;transfer
        R 100       ;Register containing count and position
        R 2000      ;source Register in slave
        DB 7999     ;destination DB in master
```

Error Handling
The "Range Error" of Diagnostic Register is set when:
- Count = 0 or > 32
- Attempt to access beyond the limit of a type of media (e.g. > R 4095)
- Data Block in the master station doesn't exist
- Data Block in the master station is defined as a Text
- Tried to get element beyond the end of the Data Block
- Tried to access a Data Block in Extension Memory (DB 4000..8191) when there is no Extension Memory in the master station

Read Data-Block size

```
SRXM     channel     ;channel number
         K 3000      ;K 3000 means "read DB size"
         DB x        ;DB whose size if to be read
         R y         ;Register number to receive the DB size
```

The return value is written to Register in the 4th operand:

0	The Data Block does not exist in the slave station
1..16384	Size of a Data Block in the slave station, in DWORDs
65535	(FFFF hex) means that the Data Block is in use as a Text in the slave station

Read the size of DB 3999 in the slave station into Register 100 in the master station using channel 2.

```
    SRXM    2
            K 3000
            DB 3999
            R 100
```

See also
SASI
STXM Transmit Media (Mode S-Bus)

Practical example for Serial-S-Bus
Inputs 0..31 are to be copied from slave station number 5 into Flags 500..531 of the master station.

Master station program

```
RECEIVE EQU PB
ERROR   EQU PB

    XOB     16
    SASI    1          ;channel 1
```

```
        100       ;definition text 100

TEXT 100 "UART:9600;"
         "MODE:SM1,R500;"
         "DIAG:F1000,R1000"

EXOB

COB     0
        0
STH     F 1002     ;if RDIA
ORH     F 1005     ;or TDIA flag = High
CPB     H ERROR    ;then handle error
STH     F 1003     ;if TBSY flag = Low
CPB     L RECEIVE  ;then read data
ECOB

PB      RECEIVE
LD      R 500      ;load station number
        5          ;(station 5)
SRXM    1          ;channel 1
        32         ;read 32 items
        I 0        ;Inputs 0..31 and copy
        F 500      ;them to Flags 500..531
EPB

PB      ERROR      ;error handler
...
EPB
```

Error handling
Testing the RDIA and TDIA diagnostic flags is optional, but recommended so that problems can be identified and the appropriate remedial action taken.
During development there may be programming errors. Other errors may be one-off communications errors caused by noise, or they may be more serious such as a broken wire.
Programming errors (Range Error, Program Error etc.) are usually recognized at the commissioning stage and can be fixed immediately.
If the NEXE flag is set, this means that the last instruction was not executed (SRXM or STXM).

Slave station program
The slave station number must be configured from Device Configurator.
For the slave station it is only necessary to assign the interface with SASI. All S-Bus communications is then handled in the background by the PCD.
It is not necessary to monitor the diagnostic flags because all communications errors are handled by the master station and do not need to be monitored by the slave.

```
XOB     16
...
SASI    1
        100

TEXT 100 "UART:9600;"
         "MODE:SS1;"
         "DIAG:F1000,R1000"
...
EXOB
```

9.27 SRXM - Recieve Media (Mode D)

Description
Reads data from the remote PCD, and copies them into destination data in the local PCD.
Transfers can be I O F to O F, R T C to R T C.
The 1st operand is the channel number.
The 2nd operand is the number of items to be transferred.
The 3rd operand is the lowest address of the source data in the remote PCD.
The 4th operand is the lowest address of the destination data in the local PCD.
The TBSY Flag is set High during the execution of SRXM, and it is set Low when the operation has completed.

Format
```
SRXM[X]   [=] channel     ;channel number
          [=] count       ;number of items to transfer 1..16
          [=] source (i)  ;source address I O F R T C
          [=] dest   (i)  ;destination address O F R T C
```

Example
```
SRXM      0      ;read the contents of R 100..115
          16     ;into R 0..15 via serial channel 0
SRXM      0      ;using channel 0
          16     ;read 16 registers
          R 100  ;from R 100..115
          R 0    ;into R 0..15
```

Flags

ACCU	Unchanged
Status Flags E	Error flag set if channel not correctly initialized, or SRXM executed while already busy

See also
SASI
STXM Transmit Media (Mode D)

Practical example
Copy Inputs 0..15 from the remote PCD to Outputs 32..47 of this PCD. The two PCDs are connected by a serial line.

This (local) PCD
```
    XOB     16
    SASI    1
            0

    TEXT 0  "UART:9600,7,E,1;MODE:MD0;"
            "DIAG:F1000,R1000;"

    EXOB

    COB     0
            0
    STH     F 1003      ;if not already busy (TBSY)
    JR      H Next
    SRXM    1           ;then receive on channel 1
            16          ;16 items
            I 0         ;from I 0..15 of remote PCD
            O 32        ;into O 32..47) of this PCD
Next:
    ECOB
```

Remote PCD: Only the serial line needs to be assigned

```
    XOB     16
    SASI    1
            0

    TEXT 0  "UART:9600,7,E,1;MODE:SD0;"
            "DIAG:F1000,R1000;"

    EXOB
```

9.28 SRXM - Receive Media (Mode MM4)

Old protocol, not supported by new PCD models.

Description

Copies the receive buffer (received frame) into consecutive Registers in the PCD.
When a telegram has been received without errors: RFUL is set to 1, SRXM resets this flag to 0.
The 1st operand is the channel number.
The 2nd operand is always 0.
The 3rd operand is a Register or a Counter which will contain (after the execution of the instruction) the number of received characters.
The 4th operand is the address of the first Register to which the received characters will be copied.
Each received character uses 8 bits of a Register, so a Register can hold a maximum of 4 characters.

The characters are placed in the Registers as follow:

```
R 1:     11111111  22222222  33333333  44444444     Characters 1 to 4
R 2:     55555555  66666666  77777777  88888888     Characters 5 to 8
…
```

If the number of received characters is not a multiple of 4, the rest of the last Register is set to 0.

The address of the station which sent the telegram is contained in the <repartner> Register defined in the SASI text.

Format

```
SRXM     [=] channel     ;channel number
         [=] 0           ;not used, always 0
         [=] reg1        ;register containing the number of characters to read
         [=] reg2        ;first address of destination registers
```

After execution, `reg1` contains the number of characters actually read. A Counter can also be used. `reg2` is the address of the first Register into which the characters will be copied. A Register holds up to 4 characters)

Example

```
SRXM     1       ;transfers the telegram received on channel 1
         0       ;(not used, always 0)
         C 100   ;Counter 100 holds the number of characters read
         R 20    ;R 20 onwards receives the characters, 4 per Register
```

Flags

ACCU	Unchanged
Status Flags E	Error flag set if channel not correctly initialized, or SRXM executed while already busy

See also
SASI
STXM Transmit Media (Mode MM4)

9.29 SRXMI - Receive Media Indirect (Mode S-Bus)

Description
This instruction works in the same way as the SRXM instruction.
The only difference is that it uses indirect mode, which means that the addresses of the data for the source and destination are supplied in a Register.
SRXMI can only transfer media (R T C I O F). Special function data, like the Real Time clock, Display Register etc. cannot be transferred.

Notes
- For firmware versions earlier than 1.20.00, the max. Register address for indirect instructions is 8191.
- To use Register addresses 8192..16383 with firmware version 1.2.00 or later, set the Build Option "Use 16-bit Register and Flag addressing" to Yes.
- This instruction cannot be used with Function Block parameters (= n).
- Temporary Registers, defined with TEQU, cannot be used.

Format
```
SRXMI    channel   ;channel number or R containing the channel number
         count     ;R containing the Count or Count + Position
         source    ;source data type and reg number with source data base adds
         dest      ;dest data type and reg number with dest data base address
```

`count`	count or count + position The number of a Register which contains the number of items to transfer (count) 1..32 in the MS word (bits 31..16), and the starting item number (position) in the LS word (bits 15..0). This Register can be loaded using LDL first to load the position, and LDH to load the count.
`source`	The data type of the source data, e.g. I O F R T C DB, and the number of the Register which contains the source address, e.g. F 123.
`dest`	The data type of the destination data, e.g. O F R T C DB, and the number of the Register which contains the destination address, e.g. O 124. The source and destination data types must be compatible.

Example
Transfer Output 200..231 (32 items) from the slave station to Flags 1000..1031 in the master station via channel 3.

```
LD      R 100     ;load the Count
        32        ;32 items
LD      R 101     ;load the source base address
        200       ;Output 200
LD      R 102     ;load the destination address
        1000      ;Flag 1000
```

```
SRXMI  3         ;channel 3
       R 100     ;count is in R 100
       O 101     ;source type O, base address in R 101
       F 102     ;destination type F, base address in R 102
```

Use the Diagnostic Flags and Diagnostic Register for detecting communications errors.

Flags

ACCU	Unchanged
Status Flags E	Error flag set if channel not correctly initialized, or SRXM executed while already busy

See also

SASI

STXMI Transmit Media indirect (Mode S-Bus)

For more information, see the "S-Bus Manual" (Ref. 26/739)

9.30 STXM - Transmit Media (Mode S-Bus)

Description

Copies data from the master station to a slave station.
The slave's station number must be loaded into the Register defined by the SASI Text.
The SASI instruction must be executed before this instruction.
This instruction can only be used by the master PCD.

The TBSY Flag is set High while it is being processed, and is reset once the data transfer is complete.
Before executing the SRXM instruction, the TBSY Flag can be tested to ensure it is Low.

The instruction has four lines:
The 1st operand is the channel number.
The 2nd operand defines the number of items to be sent.
The 3rd operand defines the base address (lowest) of the source data in the master PCD.
The 4th operand defines the base address (lowest) of the destination data in the slave PCD.

Format

```
STXM[X]   [=] channel     ;channel number or register containing the channel numbe
          [=] count       ;number of items to transmit
          [=] source (i)  ;base address of source data (in master)
          [=] dest   (i)  ;base address of destination data (in slave)
```

`count`	`1..32`	Number of R T C to transmit
	`1..128`	Number of I O F to transmit
	`0`	Special function code, see below
`source`	Base address of data in the master PCD:	
	`I O F`	Input, Output, Flag
	`R`	Register
	`T C`	Timer, Counter
	`DB`	Data Block
	`K`	K constant for special function
`dest`	Base address of data in the slave PCD:	
	`I O F`	Input, Output, Flag
	`R`	Register
	`T C`	Timer, Counter
	`DB`	Data Block
	`K`	1000, Write clock in the slave PCD
	`K`	17, 18, 19: Special functions, see below

<u>Data source and destination</u>
The following table shows which data can be copied from the slave station to the appropriate data in the master station.

	Master PCD (destination)						
		O	F	R	C	T	DB
	I	x	x				
	O	x	x				
Slave PCD	F	x	x				
(source)	R			x	x	x	x
	C			x	x	x	x
	T			x	x	x	x
	K			x			
	DB			x	x	x	

When writing to the clock, two Registers are sent. For the data format of the Registers, see the WTIME instruction.

<u>Example</u>
Copy I O F R T C values to a slave station:

```
LD      R 100      ;register as defined in the SASI text
        22         ;with the slave station number 22
STXM    0          ;transmits via channel 0
        100        ;100 items
        F 100      ;from F 100..199
        O 32       ;to O 32..131 of station 22
```

Special function to execute an XOB in a slave station:

```
LD      R 100     ;register as defined in the SASI text
        12        ;with the slave station number 12
STXM    1         ;channel number 0
        0         ;(must be 0)
        K 4000    ;special function number for XOB interrupt
        K 17      ;number of the XOB to execute (17 | 18 | 19)
```

Special function to copy the master clock in a slave station:

```
LD      R 100     ;register as defined in the SASI text
        255       ;with broadcast mode 255 = all slaves
RTIME   R 20      ;copy Master clock to two registers 20 and 21
STXM    1         ;copy master clock from R 20..21 to slave stn clock
        0         ;(must be 0)
        R 20
        K 1000    ;K 1000 = clock
```

Broadcast mode

It is also possible to use the STXM instruction in broadcast mode. The slave address 255 means that all the slaves on the network receive the telegram.
This allows the synchronization of events. (Broadcast telegrams are not supported with SRXM instructions.)

Flags

ACCU	Unchanged
Status Flags E	Error flag set if channel not correctly initialized, or STXM executed while already busy

Writing Data Blocks

The format of the STXM instruction is slightly different for Data Blocks. To address a Data Block element, both the DB number and the element number are needed.

```
SRXM[X]   channel             ;channel number
          count_position      ;register with number of elements and offset
          source      (i)     ;base adds of source element (in slave) (note 1)
          destination (i)     ;base adds of destination element (in master)
```

`count+position`	The number of a Register which contains the number of items to transfer (count) 1..32 in the MS word (bits 31..16), and the starting item number (position) in the LS word (bits 15..0). This Register can be loaded using LDL first to load the position and LDH to load the count.
`source` `destination`	The source and destination addresses must be compatible data types, see the table above.

Note 1) When using STXMX in indexed mode, the source and destination are both indexed with standard media (I O F R T C),
but Data Blocks are not indexed.

Example

Transfer 20 items from Data Block 4000 position 50 in the master station to register 1000..1019 in the slave station via channel 1.

```
LDL     R 100      ;load the position in the DB
        50
LDH     R 100      ;load the count
        20
```

```
STXM   1          ;transfer via channel 1
       R 100      ;R 100 contains the count+position
       DB 4000    ;source DB
       R 1000     ;base destination register
```

See also
SASI
SRXM Receive Media (Mode S-Bus)
Diagnostic Register

Practical example for Serial-S-Bus
Registers 150..165 are to be copied from the master station to Counters 500..515 of slave station 12.

Master station program

```
    TRANSMIT EQU PB
    ERROR EQU PB

    XOB    16
    ...
    SASI   1            ;channel 1
           900          ;definition text 900

    TEXT 900 "UART:9600;"
             "MODE:SM1,R500;"
             "DIAG:F2500,R4095"

    EXOB

    COB    0
           0
    ...
    STH    F 2502       ;if RDIA
    ORH    F 2505       ;or TDIA Flag = High
    CPB    H ERROR      ;then handle error
    STH    F 2503       ;if TBSY Flag = Low
    CPB    L TRANSMIT   ;then transmit data
    ...
    ECOB

    PB     TRANSMIT
    LD     R 500        ;load station number 12
           12
    STXM   1            ;channel 1
           16           ;transmit 16 items
           R 150        ;from Registers 150..165
           C 500        ;to Counters 500..515
    EPB

    PB ERROR            ;error handler
    ...
    EPB
```

Error handling
Testing the RDIA and TDIA diagnostic flags is optional, but recommended so that problems can be identified and the appropriate remedial action taken.

During development there may be programming errors. Run-time errors may be one-off communications errors caused by noise, or they may be more serious such as a broken wire. Programming errors (Range Error, Program Error etc.) are usually recognized at the commissioning stage and can be fixed immediately.
If the NEXE flag is set, this means that the last instruction was not executed (SRXM or STXM).

Slave station program
The slave station number must be configured from Device Configurator.
For the slave station it is only necessary to assign the interface with SASI. All S-Bus communications is then handled in the background by the PCD.
It is not necessary to monitor the diagnostic flags because all communications errors are handled by the master station and do not need to be monitored by the slave.

```
    XOB     16
    ...
    SASI    1
            100

    TEXT 100 "UART:9600;"
             "MODE:SS1;"
             "DIAG:F1000,R1000"
    ...
    EXOB
```

9.31 STXM - Transmit Media (Mode D)

Description
Transmits date from the local PCD to data in the Remote PCD.
The data can be I O F to O F, or R T C to R T C.
The 1st operand is the channel number.
The 2nd operand is the number of items to be transferred.
The 3rd operand is the lowest address of the source data in the local PCD.
The 4th operand is the lowest address of the destination data in the remote PCD.

The TBSY Flag is set High during the execution of STXM, and it is set Low when the operation has completed.

Format
```
STXM[X]   [=] channel     ;channel number
          [=] count       ;number of items to transmit 1..16
          [=] source (i)  ;source address I O F R T C
          [=] dest   (i)  ;destination address O F R T C
```

Example
```
STXM    0       ;transmits the contents of R 100..115
        16      ;into R 0..15 via serial channel 0
        R 100
        R 0
```

Flags

ACCU	Unchanged
Status Flags E	Error flag set if the channel has not been correctly initialized or does not exist, or is already transmitting.

See also
SASI

SRXM Recieve Media (Mode D)

Practical example
Copy Inputs 0..15 of the local PCD to the Outputs 32..47 of the remote PCD.

Program in local PCD

```
    XOB     16
    SASI    1
            15

    TEXT 15 "UART:9600,7,E,1;MODE:MD0;DIAG:F1000,R1000;"

    EXOB

    COB     0
            0
    STH     F 1003      ;if not already busy (TBSY)
    JR      H Next
    STXM    1           ;then transfer on serial channel 1
            16          ;16 items
            I 0         ;from Inputs 0..15 of local PCD
            O 32        ;to Outputs 32..47 of remote PCD
Next:
    ECOB
```

Remote PCD
Only the serial channel need to be assigned, see SRXM.

9.32 STXM - Transmit Media (Mode MM4)

Old protocol, not supported by new PCD models.

Description
Transfers Registers over the LAC/LAC2 network using the MM4 protocol.
This transfer can occur via a LAC/LAC2 network or point-to-point.
The 1st operand is the channel number.
The 2nd operand defines the transfer function.
The 3rd operand is a Register or a Counter which contains the number of characters to transfer.
The 4th operand is the address of the first Register containing the characters to transmit.

A Register can hold a maximum of 4 characters: each character needs 8 bits.
The characters must be loaded into the Registers as follows:

R 1:	11111111	22222222	33333333	44444444	Characters 1 to 4
R 2:	55555555	66666666	77777777	88888888	Characters 5 to 8

...

The address of the partner is contained in the <trpartner> Register defined in the SASI Text.

The TBSY Flag is set High during the execution of STXM, and it is set Low when the operation has completed.

Format

```
STXM    [=] channel       ;channel number
        [=] fcn           ;function to perform 0..4
        [=] reg1          ;Register containing the number of characters to transmit
        [=] reg2         ;base address of source Register
```

`fct`		Function to perform
	0 /2	Transmission of data
	4	Broadcast
`reg1`		Register containing the number of characters to be transmitted (a Counter can also be used).
`reg2`		Address of the first Register from where the information is to be transferred (a Register holds up to 4 characters)

Example

```
STXM    1       ;transmit on channel 1
        0       ;indicates a transmission
        C 100   ;number of characters to transmit in Counter 100
        R 20    ;1st Register containing the data
```

Flags

ACCU		Unchanged
Status Flags	E	Error flag set if the channel has not been correctly initialized or does not exist, or is already transmitting.

See also

SASI
SRXM Receive Media (Mode MM4)

9.33 STXMI - Transmit Media Indirect (Mode S-Bus)

Description

This instruction works in the same way as the STXM instruction.
The only difference is that it uses indirect mode, which means that the addresses of the data for the source and destination are supplied in a Register.
STXMI can only transfer media (R T C I O F). Special function data, like the Real Time clock, Display Register etc. cannot be transferred.

Notes

- For firmware versions earlier than 1.20.00, the max. Register address for indirect instructions is 8191.
- To use Register addresses 8192..16383 with firmware version 1.2.00 or later, set the Build Option "Use 16-bit Register and Flag addressing" to Yes.
- This instruction cannot be used with Function Block parameters (= n).
- Temporary Registers, defined with TEQU, cannot be used.

Format

```
STXMI    channel   ;channel number or reg containing channel number
         count     ;reg containing the Count or Count + Position
         source    ;source data type and reg number with source data base address
         dest      ;dest data type and reg number containing dest base address
```

`count`	count or count + position The number of a Register which contains the number of items to transfer (count) 1..32 in the MS word (bits 31..16), and the starting item number (position) in the LS word (bits 15..0). This Register can be loaded using LDL first to load the position, and LDH to load the count.
`source`	The data type of the source data, e.g. I O F R T C DB, and the number of the Register

	which contains the source address, e.g. F 123.
`dest`	The data type of the destination data, e.g. O F R T C DB, and the number of the Register which contains the destination address, e.g. O 124. The source and destination data types must be compatible.

Example

Transfer 20 values from DB 4000 positions 50..69 in the master station to Registers 1000..1019 in the slave station via channel 1.

```
LDL     R 100     ;load the DB position
        50
LDH     R 100     ;load the Count (number of values)
        20
LD      R 101     ;load the source DB number
        4000
LD      R 102     ;load the destination register number
        1000
STXMI   1         ;channel number 1
        R 100     ;Count + Position: MSW of R 100 = 20; LSW of R 100 = 50
        DB 101    ;R 101 = 4000
        R 102     ;R 102 = 1000
```

Flags

ACCU	Unchanged
Status Flags E	Error flag set if the channel has not been correctly initialized or does not exist, or is already transmitting.

See also

SASII

SRXMI Receive Media indirect (Mode S-Bus)

9.34 SICL - Serial Input Control Line

Description

Reads a control signal from the serial channel and stores its state in the ACCU.

Format

```
SICL     [=] channel     ;channel number
         [=] signal      ;signal number, 0=CTS, 1=DSR, 2=DCD
```

`signal`	0 = CTS	Clear To Send
	1 = DSR	Data Set Ready
	2 = DCD	Data Carrier Detect

Example

```
CTS EQU 0
DSR EQU 1
DCD EQU 2
...
SICL     3      ;if DSR of channel 1 is High
         DSR
CPB      H 25   ;then call PB 25
...
```

Flags

ACCU		Set according to the state of the assessed control line.
Status Flags	E	Set if the channel does not exist or has not been correctly initialized with a SASI instruction.

See also
SOCL Serial Control Output Control Line

Tips:
- For a channel configured for S-Bus Level 2 for Public Line Modem, the user can read the DCD signal to detect whether the PCD is on-line with a remote modem or not.
  According to the DCD state he can then execute different code in the user program.
- The connection of a programming unit can be detected by reading the DSR signal (DSR = 1).
- It is not possible to detect whether the PCD is online with S-Bus Level 2 or not since the DSR signal on the PGU port (PCD1/PCD2/ PCD4/PCD6M5/M3) is LOW for S-Bus Level 2 as well as if the port is free for any user assignation (SASI).

9.35 SOCL - Serial Output Control Line

Description
The SOCL instruction sets a selected control signal of the serial channel given in the first operand to the state of the ACCU (H or L)

Format
```
SOCL     [=] channel     ;channel number
         [=] signal      ;signal number 0=RTS 1=DTR 2=RS-232/422/485
```

`signal`	0 = RTS	Request To Send
	1 = DTR	Data Terminal Ready
	2 = RS232/422/485	Data Carrier Detect

Example

```
RTS EQU 0
...
SOCL     0      ;sets DTR signal of channel 0 according to
         RTS    ;the ACCU state
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the channel does not exist or has not been correctly initialized with a SASI instruction.

See also
SICL Serial Input Control Line

Practical examples

Port 0 on PCD2
A SASI for SM1/SS1 in the user program has configured Channel 0 to RS-485.
If the user wants to use RS-232 on the Channel 0 then the following instructions must be used (after the SASI instruction):
```
ACC     L
SOCL    0
```

```
         2
```

Switching from RS-485 to RS-422
The serial interface RS-422/RS485 switches automatically to RS-485 when certain modes are assigned.

Mode	Type
MC0..MC3, MD0 / SD0	RS-422
MC4, S-Bus	RS-485

It is sometimes needed to force the PCD to use S-Bus with RS-422.
In this case, the following instructions must be performed after the SASI instruction:

```
ACC     L
SOCL    channel
        2
```

Force the RS-485 mode with MC0..MC3 or MD0/SD0

```
ACC     H
SOCL    channel
        2
```

Switch from receive to transmit in RS-485
To set the RS-485 in the transmit mode perform the following instructions after the SASI instruction:

```
ACC     H
SOCL    channel
        0
```

To set the RS-485 in the receive mode perform the following instructions after the SASI instruction:

```
ACC     L
SOCL    channel
        0
```

9.36 SCON - Control Communication (Profibus-DP)

Description
For data exchange between PCDs on a Profibus-DP channel.
The 1st operand is the channel number.
The 2nd operand is a function code which defines the action to be taken.
The 3rd operand is a parameter dependent on the function code.

Format

```
SCON    [=] channel      ;channel number
        [=] func_code    ;function code, see below
        [=] parameter    ;parameter for the specified function 0..255
```

Example

```
SCON     9     ;Profibus-DP channel 10
         1     ;function 1=read slave diagnostic data
         4     ;slave number 4
```

Flags

ACCU		Unchanged
Status Flags	E	Set if the channel does not exist or has not been correctly assigned.

Function Codes

Function		**Parameter**	**Description**	**Diagnostic affected**	
Master	**Slave**			**Flag**	**Reg**
0		0	Stop data exchange between master and slaves	2, 3	1
1		Slave no. 0..126	Read slave diagnostic data	0, 2	3 - 6 0,7,8,9,10-69
			Start / Stop default data exchange between image memory and the PROFIBUS-DP card		
2	2	0	Stop default model data exchange for all slaves between the entire image memory and the Profibus-DP card (COB 0; ECOB)		
		1	Start default model data exchange for all slaves between the entire image memory and the Profibus-DP card (COB 0; ECOB)		
		2	Stop data exchange for all slaves between input image memory and the Profibus-DP card (Start of COB 0)		
		3	Start data exchange for all slaves between input image memory and the Profibus-DP card (Start of COB 0)		
		4	Stop data exchange for all slaves between output image memory and the Profibus-DP card (End of COB 0)		
		5	Start data exchange for all slaves between output image memory and the Profibus-DP card (Ende von COB 0)		
		6	Disable update of input media related to a DP slave having an error.		
		7	Enable update of input media even if the related DP slave is in error (default on Power ON).		
3	3	0	Force data exchange for all slaves between the entire image memory and the Profibus-DP card		
		1	Force data exchange for all slaves between input image memory and the Profibus-DP card		
		2	Force data exchange for all slaves between output image memory and the Profibus-DP card		
4		Slave no. 0..126	Force data exchange for a slave device between input image memory and the Profibus-DP card		
5		Slave no. 0..126	Force data exchange for a slave device between output image memory and the Profibus-DP card		
6		Slave no. 0..126	Force data exchange for a slave device between the entire image memory and the Profibus-DP card		
7		Slave no. 0..126	Read status of a slave		2
8		Slave no. 0..126	Deactivate slave	2	1
9		Slave no. 0..126	Activate slave	2	1
10		Group no.	Force data exchange for a group of slaves		

		0..255	between input image memory and the Profibus-DP card		
11		Group no. 0..255	Force data exchange for a group of slaves between output image memory and the Profibus-DP card		
12		Group no. 0..255	Force data exchange for a group of slaves between the entire image memory and the Profibus-DP card		
13		Group no. 0..255	FREEZE	1	0
14		Group no. 0..255	UNFREEZE	1	0
15		Group no. 0..255	SYNC	1	0
16		Group no. 0..255	UNSYNC	1	0

SCON(I) 0: stop data exchange between master and slave

This instruction can be used to stop data exchange on the Profibus-DP network.
To restart data exchange, it is necessary to execute a 'Restart Cold' on the PCD.
This instruction sets all slave Outputs to 0.
It is mainly used in XOB 0, so that slave Outputs are not left in an undefined state before powering off the master.
Diagnostic Flag +2 is set High as soon as this instruction executes, and is set Low when complete.
This instruction may only be executed when Diagnostic Flag +2 is low.

When the instruction has been executed and the status of Diagnostic Flag + 2 is Low, the result of the operation is written to Diagnostic Register + 1.
See Diagnostic Registers with Profibus-DP
Diagnostic Flag +3 shows the status of data exchange on the Profibus-DP network.

Diagnostic flag +3:	L = Data exchange on the Profibus-DP. network has stopped.
	H = Data exchange on the Profibus-DP. network is running.

Format

```
SCON      channel      ;9, 8
          func_code    ;0
          parameter    ;0 = Stop data exchange on the Profibus-DP network
```

Flags

The Error flag is set if the channel is unassigned or if the instruction has been called when diagnostic flag +2 is high.

Example

Stop data exchange on the Profibus-DP network:

```
    STH     SERV_BUSY     ;if diagnostic flag +2
    JR      H Next        ;is not High (is Low), then SCON
    SCON    9             ;Profibus-DP channel 9
            0             ;function code 0
            0             ;stop Profibus-DP
Next:
    ...
```

SCON(I) 1: Read slave diagnostic data

With this instruction the diagnostic data of the slave can be read.
Diagnostic data is mostly read when an error has been detected in the slave.
This is indicated by setting diagnostic flag +0.
The user can then identify the faulty slave by means of diagnostic registers +3 to + 6 and read the diagnostic data of that slave.
As soon as this instruction is executed, diagnostic flag +2 is set high and, when the instruction is finished, reset low.
When the instruction has been executed and the status of diagnostic flag +2 is low, the result of the operation is written to diagnostic register +1.

A description of the response code is given in section 5.2.1.2, 'Diagnostic registers with PROFIBUS-DP'.
This instruction may only be executed when the status of diagnostic flag +2 is 0.

When the instruction is finished, in diagnostic registers +3 to + 6 the relevant bit for the slave to which the instruction was addressed is set low.
The following values are stored in the diagnostic registers:

Diagnostic register +7:	Length of expanded Profibus-DP diagnostic
Diagnostic register +8:	Standard Profibus-DP diagnostic bytes 0 and 1
Diagnostic register +9:	Standard Profibus-DP diagnostic bytes 2 to 5
Diagnostic register +10:	Expanded Profibus-DP diagnostic bytes 6 to 9

etc.

A description of the response code is given in here Diagnostic Registers with PROFIBUS-DP.

Format

```
SCON    channel     ;9, 8
        func_code   ;1
        parameter   ;0..126 = Station number
```

Flags

The Error flag is set if the channel is unassigned or if the instruction has been called when diagnostic flag +2 is high.

Example

Read slave diagnostic data from slave 5:

```
    STH     SLAVE_ERR     ;if diagflag +0 = High
    ANL     SERV_BUSY     ;and no SCON is active
    JR      L Next        ;(diagflag +2 = Low), then SCON
    SCON    9             ;Profibus-DP channel 9
            1             ;function code 1 = read diagnostic data
            5             ;slave number 5
Next:
    ...
```

SCON(I) 2: Start / stop default data exchange between image memory and the Profibus-DP card

With this instruction default data exchange between the image memory and the Profibus-DP card can be started or stopped.
Default data exchange refers to the data exchange that is executed automatically when COB 0 starts up and when it ends.
This data exchange can be changed to the following function:

Parameters

0	Stop default model data exchange for all slaves between the entire image memory and the Profibus-DP card (COB 0; ECOB)
1	Start default model data exchange for all slaves between the entire image memory and the Profibus-DP card (COB 0; ECOB)
2	Stop data exchange for all slaves between input image memory and the Profibus-DP card (Start COB 0)
3	Start data exchange for all slaves between input image memory and the Profibus-DP card (Start COB 0)
4	Stop data exchange for all slaves between output image memory and the Profibus-DP card (End COB 0)
5	Start data exchange for all slaves between output image memory and the Profibus-DP card (End COB 0)

Format

```
SCON    channel     ;9, 8
        func_code   ;2
        parameter   ;0..5 = Parameter
```

Flags

The Error flag is set if the channel is unassigned.

Example

Stop data exchange for all slaves between input image memory and the Profibus-DP card (Start COB 0)

```
SCON    9      ;Profibus-DP channel 9
        2      ;func_code 2
        2      ;parameter 2
```

<u>SCON(I) 3: Force data exchange for all slaves between the image memory and the Profibus-DP card</u>

With this instruction, data exchange between the image memory of all slaves and the Profibus-DP card can at any time be forced in the user program.
This forcing can take place in the following way:

Parameters

Force data exchange for all slaves between the entire image memory and the Profibus-DP card
Force data exchange for all slaves between input image memory and the Profibus-DP card
Force data exchange for all slaves between output image memory and the Profibus-DP card

Format

```
SCON    channel    ;9, 8
        func_code  ;3
        parameter  ;0..2 = parameter
```

Flags

The Error flag is set if the channel is unassigned.

Example

Force data exchange for all slaves between the entire image memory and the Profibus-DP card

```
SCON    9    ;Profibus-DP channel 9
        3    ;function code 3
        0    ;output image memory
```

SCON(I) 4, 5, 6: Force data exchange for a slave between the image memory and the Profibus-DP card

With these instructions data exchange between the image memory of a slave and the Profibus-DP card can at any time be forced in the user program.
This forcing can take place in the following way:

Function Codes

Force data exchange for a slave between input image memory and the Profibus-DP card.
Force data exchange for a slave between output imate memory and the Profibus-DP card.
Force data exchange for a slave between the entire image memory and the Profibus-DP card.

Format

```
SCON    channel     ;9, 8
        func_code   ;4, 5, 6
        parameter   ;0..126 = Slave number
```

Flags

The Error flag is set if the channel is unassigned.

Example

Force data exchange for slaves 12 between output image memory and the Profibus-DP card.

```
SCON    9           ;Profibus-DP channel 9
        5           ;function code 5
        12          ;slave 12
```

SCON(I) 7: Read status of a slave

With this instruction the status of a slave can be read. After execution of the instruction, the slave's status is written to diagnostic register + 2.
A description of diagnostic register + 2 is given in Diagnostic registers with PROFIBUS-DP.

Format

```
SCON    channel     ;9, 8
        func_code   ;7
        parameter   ;0..126 = Slave number
```

Flags

The error flag is set if the channel is unassigned.

Example

Read status of slave 34.

```
SCON    9       ;Profibus-DP channel 9
        7       ;function code 7
        34      ;Slave 34
```

SCON(I) 8, 9: Deactivate / activate slave

With this instruction a slave can be activated or deactivated.
When the instruction is executed, diagnostic flag +2 is set high and when the instruction finishes, it is set low.
After the instruction has been executed and the status of diagnostic flag + 2 is low, the result of the operation is written to diagnostic register + 1.

A description of the response code is given in section 5.2.1.2, 'Diagnostic registers with PROFIBUS-DP'.
This instruction may only be executed if the status of diagnostic flag + 2 is 0.
The deactivation or activation of a slave is triggered by the following function codes:

Function codes

8	Deactivate slave
9	Activate slave

Format

```
SCON    channel    ;9, 8
        func_code  ;8, 9 = deactivate / activate slave
        parameter  ;0..126 = Slave number
```

Flags
The Error flag is set if the channel is unassigned or if the instruction is called when diagnostic flag + 2 is high.

Example
Deactivate slave 32.

```
    STH    SERV_BUSY     ;if diagnostic flag +2
    JR     H Next        ;is not High (is Lowe), then SCON
    SCON   9             ;Profibus-DP channel 9
           8             ;function code 8
           32            ;slave 32
Next:
    ...
```

<u>SCON(I) 10, 11, 12: Force data exchange for a group of slaves between the image memory and the Profibus-DP card</u>
With these instructions, data exchange between the image memory of one or more groups of slaves and the Profibus-DP card can at any time be forced in the user program.
Assigning a slave to a group takes place with the Profibus-DP configurator.
Profibus-DP supports the formation of a maximum of 8 groups.
These groups can be assigned as many slaves as required.
The choice of group in the SCON parameter is bit-oriented according to the following pattern:

Parameters

Bit Number	
0	Group 1
1	Group 2
2	Group 3
3	Group 4
4	Group 5
5	Group 6
6	Group 7
7	Group 8

Forcing can be applied here to more than one group at a time. This forcing can take place in the following way:

Function codes

Force data exchange for a group of slaves between input image memory and the Profibus-DP

card.
Force data exchange for a group of slaves between output image memory and the Profibus-DP card.
Force data exchange for a group of slaves between the entire image memory and the Profibus-DP card.

Format

```
SCON    channel      ;9, 8
        func_code    ;10, 11, 12
        parameter   ;0..255= Group number
```

Flags

The error flag is set if the channel is unassigned.

Example

Force data exchange for groups 1 and 2 between input image memory and the Profibus-DP card.

```
SCON    9     ;Profibus-DP channel 9
        10    ;function code 10
        3     ;Groups 1 and 2 (00000011q)
```

SCON(I) 13, 14: Global Control Service Freeze, Unfreeze

With these instructions, the 'Freeze' and 'Unfreeze' commands can be triggered for one or more groups of slaves.
The instruction is used for the purpose of input synchronization.
With the 'Freeze' instruction, the master causes a slave or group of slaves simultaneously to freeze inputs in their present state.
The slaves addressed therefore stop their inputs at exactly the same time. In the next data cycle (Data_exch) the slaves transmit the frozen inputs to the master.
Any changes at the inputs are not recognized by the slaves and are also not passed on the the master.
After the conclusion of this action, the master sends an 'Unfreeze' instruction to the group. Input changes are now sent again from the slave to the master in the normal data cycle.
It is permissible for the master, after one 'Freeze' instruction, to send further 'Freeze' instructions to the slaves.
In this case the current status of inputs is frozen each time and sent to the master in the next data cycle.

Diagnostic flag +1 is set high as soon as this instruction starts up.
When the instruction has finished, the flag is set low and the result of the operation is written to diagnostic register +0.
A description of the response code in diagnostic register +0 is given in Diagnostic Registers with Profibus-DP.

This instruction may only be executed if the status of diagnostic flag +1 is low.
Assigning a slave to a group takes place with the Profibus-DP configurator.
Profibus-DP supports the formation of a maximum of 8 groups.
These groups can be assigned as many slaves as required.
The choice of group in the SCON parameter is bit-oriented according to the following pattern:

Parameter

Bit Number	
0	Group 1
1	Group 2
2	Group 3
3	Group 4
4	Group 5
5	Group 6
6	Group 7
7	Group 8

A 'Freeze' or 'Unfreeze' instruction can be executed here on several groups simultaneously.
Fct. code to trigger 'Freeze' or 'Unfreeze' instructions:

13	Start freeze instruction.
14	Start unfreeze instruction.

Format

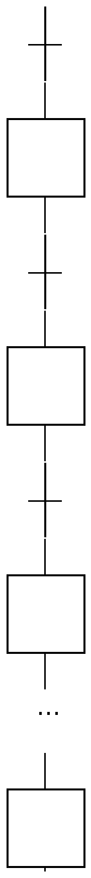
```
SCON   channel    ;9, 8
       func_code  ;13, 14
       parameter  ;0..255 = Group number
```

Flags

The error flag is set if the channel is unassigned or if the instruction is called when diagnostic flag +1 is high.

Example

Execute freeze and unfreeze sequence for the slaves of group 5.



```
STL      GCS_BUSY       ;If diagnostic flag +1
                        ; is low, then continue

SCON     9              ;Profibus-DP channel 9
         13             ; Freeze
         16             ; Group 5 (00010000)

STL      GCS_BUSY       ;If diagnostic flag +1
                        ; is low, then continue

LD       T   3          ;Load timer with
         100            ; value 100, delay so that
                        ; the slaves transmit their
                        ; frozen inputs to the
                        ; master
STL      T   3

STL      F   XX         ;Process the
                        ; frozen I/Os of slaves



SCON     9              ;Profibus-DP channel 9
         14             ; Unfreeze
16                      ; Group 5 (00010000)
```

```
            STL      GCS_BUSY       ;If diagnostic flag +1
                                    ; is low, then continue
```

SCON(I) 15, 16: Global Control Service Sync, Unsync
With these instructions, the 'Sync' and 'Unsync' commands can be triggered for one or more groups of slaves.
The instruction is used to synchronize the outputs.

With the 'Sync' instruction, the master causes a slave or group of slaves simultaneously to freeze outputs in their present state. In the next data cycle (Data_exch) the master transfers the output image to the slaves, without the slaves copying this image to their outputs. After the conclusion of this action, the master sends an 'Unsync' instruction to the group. All slave outputs are now switched on or off at precisely the same time and these outputs are again refreshed in the normal data cycle. It is permissible for the master, after one 'Sync' instruction, to send further 'Sync' instructions to the slaves. In each case the current output image is copied to the outputs at exactly the same time.

Diagnostic flag +1 is set high as soon as this instruction starts up. When the instruction has finished, the flag is set low and the result of the operation is written to diagnostic register +0. A description of the response code in diagnostic register +0 is given in section 5.2.1.2, 'Diagnostic registers with PROFIBUS-DP'. This instruction may only be executed when the status of diagnostic flag +1 is low.

Assigning a slave to a group takes place with the Profibus-DP configurator. Profibus-DP supports the formation of a maximum of 8 groups. These groups can be assigned as many slaves as required. The choice of group in the SCON parameter is bit-oriented according to the following pattern:

Parameter

Bit Number	
0	Group 1
1	Group 2
2	Group 3
3	Group 4
4	Group 5
5	Group 6
6	Group 7
7	Group 8

A 'Sync' or 'Unsync' instruction can be executed here on several groups simultaneously.

Function codes
to trigger 'Sync' or 'Unsync' instructions:
Start sync instruction.

Start unsync instruction.

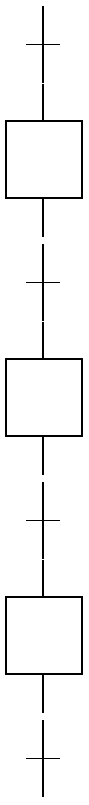
Format

```
SCON   channel    ;9, 8
       func_code  ;15, 16
       parameter  ;0..255 = Group number
```

Flags
The Error flag is set if the channel is unassigned or if the instruction is called when diagnostic flag +1 is high.

Example

Execute a 'Sync' and 'Unsync' sequence for the slaves of group 3.



```
STL     GCS_BUSY      ;If diagnostic flag +1
                      ; is low, then continue

SCON    9             ;Profibus-DP channel 9
        15            ; Sync
        4             ; Group 3 (00000100)

STL     GCS_BUSY      ;If diagnostic flag +1
                      ; is low, then continue

OUT     F   XX        ;Set outputs
                      ; of slaves
LD      T   5         ;Load timer 5 with
        400           ; value 400

STL     T   5         ;Wait until timer = 0

SCON    9             ;Profibus-DP channel 9
        16            ; Unsync
        4             ; Group 3 (00000100)

STL     GCS_BUSY      ;If diagnostic flag +1
                      ; is low, then continue
```

History List Messages

In case of problems with Profibus-DP the following error message is stored in the history log:
PROF DP FAIL xxx

ERR#	Description
0	Keyword MODE: not found
0	Wrong mode specified
0	Keyword CONF: not found
0	DBX key word not specified
0	DBX number error
0	DBX number to large
0	DBX does not exist
0	Keyword DIAG: not found
0	Flag or output key word not specified in DIAG
0	Error in address of diag flag or output
0	Range error diag flag or output
0	Register keyword not specified in DIAG
0	Range error diag register
1	Profibus-DP HW card not present
2	Error in instruction
3	DBX structure error

4	DBX type not for DP master (no PROFIBUS DBX)
5	FW-DBX version not compatible
6	No IN RING message after timeout on initialization
7	Semaphore error for data exchange (info to PCD support)
8	DBX error: data transfer function not implemented
9	Incompatible PCD7.F750 and PCD hardware

See also
For more information, consult the "Profibus-DP Manual"

9.37 SCONI - Control Communication Indirect (Profibus-DP)

Description
Controls data exchange between PCDs on a Profibus-DP channel. This is the same as SCON, but the operands are passed in Registers.

Notes
- This instruction cannot be used with Function Block parameters (= n).
- Temporary Registers, defined with TEQU, cannot be used.

Format
```
SCONI    reg_chan    ;channel number or reg containing channel number
         reg_func    ;register containing function code
         reg_param   ;register containing parameter
```

Example
```
SCONI    R 100    ;channel from R 100
         R 101    ;function from R 101
         R 102    ;parameter from R 102
```

Flags

ACCU		Unchanged
Status Flags	E	Error flag set if the channel has not been correctly initialized or does not exist.

Practical example
Data is to be exchanged between the PCD controller's process image memory and that of the Profibus-DP card, controlled by the user program.

```
    LD     R 2000     ;load Register 2000
           9          ;with channel 9
    LD     R 2001     ;load Register 2001
           3          ;with function code 3 = Force data exchange
    LD     R 2002     ;load Register 2002
           0          ;with parameter 0 =
    SCONI  R 2000     ;transfer process image memory
           R 2001     ;function code in R 2001
           R 2002     ;parameter in R 2002
```

See also
SCON Control Communication (Profibus-DP)
For more information, search for **Profibus-DP** in the SBC website http://www.sbc-support.com.

10 Control Instructions

These instructions control the execution of the program.

JR	Jump Relative
JPD	Jump Direct
JPI	Jump Indirect
HALT	Halts the CPU
LOCK	Lock Semaphore
UNLOCK	Unlock Semaphore

NOTE
Jump instructions are common causes of errors (infinite loops etc.) and should be used with care. COBs should not contain code which causes long loops - COBs are cyclic tasks.

10.1 JR - Jump Relative

Description
Conditionally or unconditionally jumps a specified number of program lines forwards or backwards from the current program line number.

The number of lines that can be jumped is 4095 (backwards) to +4095 (forwards), the program line jumped to is calculated by adding this value to the number of the program line containing the JR instruction. It is illegal to jump out of the current block (COB, PB, FB, ST, TR or SB): the destination MUST be in the current block.

The following condition codes can be used:

blank	Unconditional jump (condition code blank)
H	Jump if Accumulator = H (1)
L	Jump if Accumulator = L (0)
P	Jump if Positive flag = H (Negative flag = L)
N	Jump if Negative flag = H
Z	Jump if Zero flag = H
E	Jump if Error flag = H

If the condition is not true, the jump is not made; execution continues with the instruction following JR.

It is usual to use Labels (symbolic names) for jump destinations.

Format
```
JR    [cc] offset    ;cc = condition code, H L P Z N E
                     ;offset is the relative number of lines
                     ;to be jumped (4095.. +4095)
```

Example
```
    JR    H -2      ;jump 2 line above
    ...
    JR    H Repeat  ;jump to label "Repeat"
    ...
Repeat:             ;label
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
JPD
JPI
LD

Tip: Avoid creating program loops using jumps. Loops can slow the operation of COBs (tasks) within the PCD.
Instead, use linkages based on the ACCU, or consider using Graftec.
Or if a condition is not satisfied, continue processing other conditions.

Practical example

```
    STH     I 15               STH      I 15
    ANL     XBSY               ANL     XBSY
    DYN     F 15               CPB     H 25
    JR      H Next     ----->  ...
    STXT    1
            57                 PB      25
    ...                        STXT    1
Next:                                  57
    ...                        EPB
```

10.2 JPD - Jump Direct

Description
Jumps conditionally or unconditionally to a program line number relative to the start of the current block (COB, XOB, PB, FB, ST or TR).
The destination line number is always positive, between 0 and the number of lines in the current block (max 8191 lines).
Labels can also be used.

The following condition codes can be used:

blank	Unconditional jump (condition code blank)
H	Jump if Accumulator = H (1)
L	Jump if Accumulator = L (0)
P	Jump if Positive flag = H (Negative flag = L)
N	Jump if Negative flag = H
Z	Jump if Zero flag = H
E	Jump if Error flag = H

If the condition is not true, the jump is not made; execution continues with the instruction following JPD.

Format
```
JPD    [cc] offset    ;cc = condition code, H L P Z N E
                      ;offset from start of block (0..8191)
```

Example
```
JPD    L 10    ;if the ACCU is Low, a jump is made
               ;to 10th line of the current block
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
JR
JPI

10.3 JPI - Jump Indirect

Description
Similar to JPD: jumps conditionally or unconditionally to a program line number relative to the start of the current block (COB, XOB, PB, FB, ST or TR).
The program line number is read from the given Register number (only the least 13 bits are significant). Since this instruction utilizes a condition code, the 'R' data type code is omitted.
This useful for creating jump tables and 'case' statements.

Notes
- The destination of the jump cannot be outside the current block.
- For firmware versions earlier than 1.20.00, the max. Register address for indirect instructions is 8191.
- To use Register addresses 8192..16383 with firmware version 1.2.00 or later, set the Build Option "Use 16-bit Register and Flag addressing" to Yes.
- Temporary Registers, defined with TEQU, cannot be used.
- This instruction cannot be used with Function Block parameters (= n).

The following condition codes can be used:

blank	Unconditional jump (condition code blank)
H	Jump if Accumulator = H (1)
L	Jump if Accumulator = L (0)
P	Jump if Positive flag = H (Negative flag = L)
N	Jump if Negative flag = H
Z	Jump if Zero flag = H
E	Jump if Error flag = H

If the condition is not true, the jump is not made; execution continues with the instruction following JPI.
The value of a label can be loaded into a Register using the LD instruction.

Format
```
JPI      [cc] reg     ;cc = condition code, H L P Z N E
                      ;reg = Register (max. 8191) containing
                      ;the offset from start of block (0..8191)
```

Example
```
JPI      H 300     ;if the ACCU is High, a jump is made
                   ;to the line of the current block stored in
                   ;Register 300
```
Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
JR
JPD

LD

10.4 HALT - Halts Program Execution

Description
Conditionally or unconditionally Halts the PCD. If the condition is not true, the HALT is not made and execution continues with the following instruction.

The Halt state is not the same as the Stop state. After a HALT the PCD can only be set to Run by a Restart operation, or by powering the PCD off and on.

The state of the Outputs after the HALT is defined by jumpers in old PCD types, or from the Device Configurator for new models.

Use HALT only for processing fatal non-recoverable errors.

The following condition codes can be used:

blank	Unconditional jump (condition code blank)
H	Halt if Accumulator = H (1)
L	Halt if Accumulator = L (0)
P	Halt if Positive flag = H (Negative flag = L)
N	Halt if Negative flag = H
Z	Halt if Zero flag = H
E	Halt if Error flag = H

Format
```
HALT    [cc]    ;cc = condition code, H L P Z N E
```

Example
```
HALT    E       ;halt if the Error (E) flag is set
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

Practical example
If the Error (E) flag is set, diagnostic information is stored and the PCD halts.

```
    XOB     13
    DIAG    R 1000
    HALT
    EXOB
```

11 Definition Instructions

These instructions are executed on power up, and are executed ONCE only. If an instruction is executed again it is ignored.
Normally these instructions will be placed in the start-up XOB 16.
The operands of these instructions cannot be supplied as Function Block parameters.

DEFVM	Define Volatile Memory (Flags)
DEFTC	Define Timers/Counters
DEFTB	Define Timebase
DEFTR	Define Timer Resolution
DEFTMP	Define Temporary Data Size

11.1 DEFVM - Define Volatile Memory (Flags)

Description
Defines the area of Flags which are to be nonvolatile (battery-backed). Nonvolatile Flags retain their values even after power to the PCD is lost.
Volatile Flags are all set to 0 on power-up of the PCD. All Flags from the Flag address in the operand upwards are defined as being non-volatile.

If the instruction is not used, all Flags are non-volatile by default.

Note
The PG5 generates this instruction from the Project Manager's 'Build Options' as part of the build process, so you do not normally need to use the DEFVM instruction in your program.
If you have imported an old PG3 project, just delete the DEFVM instruction and set the number of nonvolatile flags in the 'Build Options'.

Format
```
DEFVM   flag    ;flag = volatile/nonvolatile flag partition 0..8191
```

Example
```
DEFVM   200     ;Flags 0  199 are volatile (set to zero on reset)
                ;Flags 200..8191 are nonvolatile (unchanged by reset)
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
DEFTC
DEFTB

Practical example
Flags 0..199 are to be declared to be volatile (set to zero on start-up). Flags 200..8191 are non-volatile (battery-backed).

```
    XOB    16     ;cold start XOB
    DEFVM  200    ;Flags 200..8191 are nonvolatile
    ...
    EXOB
```

11.2 DEFTC - Define Timers/Counters

Description

Defines the number of Timers for the PCD. Timers and Counters occupy the same addressing space. All elements BELOW the operand value are Timers, all the others are Counters.

If the instruction is not used, the default is:

Timers:	0 31
Counters:	32 1599.

Note

The PG5 generates this instruction from the Project Manager's 'Build Options' as part of the build process.
You do not need to use a DEFTC instruction in your program, instead define the number of Timers from the 'Build Options'.
If you have imported an old PG3 project, just delete the DEFTC instruction and set the number of Timers as described above.

The new SYSCMP instruction can also be used to create Timers with 1ms accuracy.

Format

```
DEFTC   ctr     ;lower limit for Counters, 0..450
```

Example

```
DEFTC   64      ;Timers are 0..63, Counters are 64..1599
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also

DEFTB
DEFTR
DEFVM
SYSCMP

Practical example

Assume that 100 Timers are necessary for an application.

```
    XOB     16      ;cold start XOB
    DEFTC   100     ;0..99 are Timers
                    ;100..1599 are Counters
    ...
    EXOB
```

11.3 DEFTB - Define Timebase

Description

Defines the timebase for the decrementing of the Timers. The operand indicates the timebase in 10's of milliseconds.
Values of 1 to 1000 are valid (10 ms to 10 sec).
If the timebase is not defined (no DEFTB), the default is 100 ms (1/10 sec).

If you load a timer with a certain time, then the time loaded into the timer depends on the value off the system timebase, for example:

```
    LD   T 33
```

```
          5
```

With a timebase of 10ms, the time loaded into the timer 33 will be 5 * 10ms = 50 ms.
If you now change the timebase to 1000ms then the timer will be loaded with 5s.

In most of the cases you would like to have a time loaded to the timer that doesn't depend on the Build Options setting.
You can do this by using the TIME data type:

```
    LD  T 33
        t#5s
```

Now the timer 33 always is loaded with 5s. It doesn't depend on the timebase value anymore.

Note
The PG5 generates this instruction from the Project Manager's 'Build Options' as part of the build process.
You do not need to use a DEFTB instruction in your program, instead define the Timer Timebase from the 'Build Options'.
If you have imported an old PG3 project, just delete the DEFTB instruction and set the timer timebase as described above.

Format

```
DEFTB   timebase  ;timebase in 10's of milliseconds, 1..10000
```

Example

```
DEFTB   100       ;timebase = 1 sec (100 * 10ms)
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
DEFVM
DEFTC
SETD
RESD

Practical example
Set the timebase to 1 second, for a very slow process!

```
    XOB     16     ;cold start XOB
    DEFTB   100    ;timebase is 100 * 10 ms = 1000 ms
    ...
    EXOB
```

Tips

What happens with Fupla files?
Fupla still works with 100ms as the timebase even if the IL program uses a timebase other than 100ms.
You can have fast timers (timebase = 10ms) in the IL (Instruction List) file and a Fupla file in the same project without running into problems.
Fupla does not support time declarations like: `t#5`

What happens if I load a time smaller than the timebase?
In this case the Timer will be loaded with the timebase.
Example: Timebase = 1s

```
    LD   T 54
```

```
          t#273ms
```
The time is less that 1s so the Timer will be loaded with 1 second.

What happens if I load a time with a resolution smaller than the timebase?
Then you are a very naughty boy and don't deserve any Christmas presents.
But seriously folks...
Example: Timebase = 1s
```
   LD   T 54
        T#1100ms
```
In this case Timer 54 will be loaded with a time of 1000ms only.
This is because the timebase is 1000ms, and the 100ms part of the 1100ms timer value is discarded.

11.4 DEFTR - Define Timer Resolution

Description
Defines the speed in milliseconds with which Timers will be decremented.
For example, if a "DEFTR 100" is specified, all non-zero Timers will be decremented by 100 every 100ms.
A "DEFTR 1000" will decrement all Timers by 1000 every 1000ms and so on.
If DEFTR and DEFTB are used in the same program, the message “DOUBLE TIME BASE” will appear in the History List and the PCD will automatically put itself in "HALT" upon a restart cold or on power-up.

The advantage of the DEFTR instruction (over the DEFTB) is that the values you specify when using Timers are independent of the timebase or resolution and always introduced in multiples of 10ms.
The DEFTR instruction allows a maximum Timer resolution of 10ms which means that the value specified in the instructions is rounded if necessary .

Example: DEFTR 25: a time base of 20ms will be set (25 rounded down to 20). The DEFTR instruction, as with the DEFTB instruction, also acts on the instructions SETD, RESD and OUTD.
If the DEFTR instruction is present in the user program then the time base of these instructions is fixed to 10ms independent of the specified value by DEFTB.

Note
The PG5 generates this instruction from the Project Manager's 'Build Options' as part of the build process.
You do not need to use a DEFTR instruction in your program, instead define the Timer resolution from the Build Options.

Format
```
DEFTR   resolution    ;resolution x 10 ms
```

Example
```
DEFTR   100    ;Timer resolution  = 100 msec
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
DEFTB

Practical example
The Output 20 will be set 150ms (15 * 10ms) after the instruction has been executed.

```
    XOB     16
```

```
DEFTR   200
...
EXOB

COB     0
        0
SETD    O 20
        15
...
ECOB
```

11.5 DEFTMP - Define Temporary Data Size

Description
When temporary data is defined using TEQU, S-Asm counts the number of temporary Registers and Flags used in each block, and generates DEFTMP R and/or DEFTMP F instructions to define the amount of temporary data used by the block. These instructions are inserted at the very end of the block.

When the block runs, the firmware assigns the correct amount of temporary data and initializes it to zeros.

Note
Each COB and XOB (each task) needs its own memory for temporary data. This must be defined by the instruction "`DEFTMP M` *`kbytes`*" in each COB or XOB. If temporary data is used in the COB or XOB, S-Asm will generate this instruction automatically with a default size of 2 KB. But if the COB or XOB itself does not use temporary data, but some of the called blocks do use it, then you must manually insert the "`DEFTMP M` *`kbytes`*" instruction into the COB or XOB. If this instruction is not there, the PCD will Halt with a TEMPDATA ILLEGAL error.

Format
```
DEFTMP R|F|M count     ;count is the number of temporary Registers or Flags,
                       ;or M is the size of temp memory in KB
```
Example
```
PB     0
;Declare temporary data, 2 Registers and 2 Flags
TempR1 TEQU R
TempR2 TEQU R
TempF1 TEQU F
TempF2 TEQU F
       ...        ;use the temp data
EPB
```

For the above example, the assembler generates these instructions and inserts them at the end of the block:
```
...
DEFTMP R 2   ;number of temporary Registers, inserted by S-Asm
DEFTMP F 2   ;number of temporary Flags, inserted by S-Asm
EPB
```

If this PB is called from a COB, the COB must define the total temporary data memory size for the task. If the COB itself uses temporary data, this instruction will be generated automatically by S-Asm. If the COB does not use temporary data, but one if its called blocks does, then you must enter the DEFTMP M instruction manually, see Note above.

```
COB    0
       0
...
CPB    0
...
DEFTMP M 2  ;defines 2K bytes of memory for the temp data stack
ECOB
```

See also

TEQU

12 Special Instructions

These instructions perform miscellaneous operations.

NOP	No Operation
RTIME	Read Time
WTIME	Write Time
DSP	Load Display Register
PID	P.I.D. Control
TEST	Test Hardware
DIAG	Read XOB Diagnostic
SYSRD	System Read
SYSWR	System Write
SYSCMP	System Compare
CSF	Call System Function
RDP	Read Peripheral
WRP	Write Peripheral

Note
The following instructions are not supported by the new PCD models (NT systems), PCD3, PCD2. M480 etc.

These two instructions work only with the analogue card PCA2.W1x.
To read or write values to analogue cards PCD2, PCD4 and PCD6, consult the appropriate hardware manual.

ALGI	Analogue Input
ALGO	Analogue Output

These instructions were used for accessing slow I/O modules such as the old PCA2.W2x / W3x.

STHS	Start High Slow
OUTS	Out Slow

12.1 ALGI - Analogue Input

Description
Reads a 12-bit value from a PCA2.W1x analogue module, and stores it in the specified Register.
The 1st operand contains both the A/D channel number (07) and the base address of the module.
The 2nd operand is the destination Register number.

If the first operand is supplied as an FB parameter, both the A/D channel number and the base address must be supplied on the same line.

Note
This instruction is not supported by new PCD types (NT systems, PCD3, PCD2.M480 etc). This instruction cannot be used for PCD4.Wxxx and PCD6.Wxxx modules (see the respective hardware manuals).

Format
```
ALGI[X]   [=] chan base    ;channel and base address
          [=] reg (i)      ;destination Register R
```

Example
```
ALGI    2 64  ;read analogue value from channel 2, at module base address 64
        R 10  ;and save it in R 10
```

Flags

ACCU		Unchanged
Status Flags	E	Always set Low
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
ALGO

12.2 ALGO - Analogue Output

Description
Outputs a 12-bit binary value from the specified Register to a PCA2.W1x analogue module.
The 1st operand is the Register to be output.
The 2nd operand contains both the D/A channel number, and the base address of the module.

If the second operand is supplied as an FB parameter, both the D/A channel number and the base address must be supplied on the same line.

Note
This instruction is not supported by new PCD types (NT systems, PCD3, PCD2.M480 etc). This instruction cannot be used for PCD4.Wxxx and PCD6.Wxxx modules (see the respective hardware manuals).

Format
```
ALGO[X]   [=] reg (i)    ;source register R
          [=] chan base  ;channel and base address
```

Example
```
ALGO    R 100    ;outputs the value in R 100
        3 128    ;to channel 3 of module at base I/O address 128
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
ALGI

12.3 CSF - Call System Function

Description
Conditionally or unconditionally calls a System Function, which is a function in the firmware, or code in a library.

The easiest way to call a System Function is to use the IL Editor (S-Edit), System Functions are shown in S-Edit's 'Selector Window', from where function call can be drag-and-dropped into the code. This adds the skeleton code for the call, and adds the required $include file to the program.

System Function calls can also be coded manually - examine the .LIB files for the SF library.

Each System Function library has a unique "library number", and each function in the library has a "function number". Each function can also have optional parameters.
These numbers are defined in the library's $include file, along with the function's parameters.

Usage
```
CSF     [cc] lib_number   ;library number
        func_number       ;function number to call
        [parms...]        ;optional parameters
```

Example
```
CSF     L 10     ;call library 10 if ACCU = Low
        3        ;function 3
        F 10     ;parameters for function 3
        R 32
```

Flags

ACCU		Unchanged
Status Flags	E	The Error (E) flag is set if the library or function does not exist.

See also
Condition codes

12.4 DIAG - Read XOB Diagnostic

Description
Fills a 12-Register block with diagnostic information relating to the last or the present Exception Organization Block (XOB) executed.
The operand is the lowest Register number of the block of 12 Registers.
DIAG is normally used within an XOB.

Register block usage:

Register		
0	XOB Number	Number of last or present XOB
+1	Program Line	Program Line when XOB was called
+2	Index Register	Value of Index Register when called
+3	COB Program Line	Program Line of Level 0 call
+4	Nesting Level 1 Program Line	Program Line of Level 1 call
+5	Nesting Level 2 Program Line	Program Line of Level 2 call
+6	Nesting Level 3 Program Line	Program Line of Level 3 call
+7	Nesting Level 4 Program Line	Program Line of Level 4 call
+8	Nesting Level 5 Program Line	Program Line of Level 5 call
+9	Nesting Level 6 Program Line	Program Line of Level 6 call
+10	Nesting Level 7 Program Line	Program Line of Level 7 call
+11	Not Used	Reserved

The program line numbers of the block calls (Nesting level information) give the program line where the previous call (CFB, CPB, etc) took place.
From these, it can be established exactly where the program was when the XOB was executed.

The operand cannot be supplied as a Function Block parameter.

Format
```
DIAG   reg      ;lowest address of 12 Registers, R 0 .. rmax-12
```

Example
```
DIAG   R 1000   ;store diagnostic information in
                ;Registers R 1000..1011
```

Flags
Unchanged.

Practical example
The address of the line where an error occurs must be printed.

```
    XOB     13
    DIAG    R 1000
    STXT    1 100

    TEXT 100 "$D $ H :"
             " Error Flag set at address $R1001<CR><LF>"

    EXOB
```

12.5 EXTB/EXTW - Sign Extension

Description
Converts a signed BYTE or WORD in a Register into a signed DWORD by extending the sign bits.
Indexing can be used.
This can be useful if a BYTE or WORD has been loaded into a Register by using XLD Load Data or MOV Move Data.

Format
```
EXTB[X]  R reg  (i)  ;sign-extends the BYTE in the register
EXTW[X]  R reg  (i)  ;sign-extends the WORD in the register
```

Examples
```
;R 100 contains the WORD value -1 (0FFFFh)
;extend this to a DWORD value: 0FFFFh -> 0FFFFFFFFh
EXTW  R 100

;R 200 contains the BYTE value 80h (-128)
;extend this to a DWORD value:  80h -> 0FFFFFF80h (-128)
EXTB  R 200

;R 300 contains the hex value 012345678h
;extending the LS WORD produces 00005678h, the MS WORD is set to zero
EXTW  R 300
```

Flags

ACCU		Unchanged
Status Flags	E	Unchanged
	P	Set according to the result
	Z	Set according to the result
	N	Set according to the result

See also
[XLD Load Data]
[MOV Move Data]

12.6 LOCK - Lock Semaphore

This instruction is only needed in the old PCD4 and PCD6 which could have more than one CPU that could share the same data.
It is not needed in PCDs with only one CPU.

Description
LOCK in conjunction with UNLOCK, is used to prevent access conflicts when several CPUs read or write the same elements.
100 Semaphores (special flags) are available (099).
The LOCK instruction checks the Semaphore. If it is High (another CPU has executed a LOCK), then the ACCU is set Low.
If it is Low, the ACCU and the Semaphore are set High.

It is the programmers responsibility to ensure that the CPU does not reference an element if the associated Semaphore is High (ACCU = L (0) after LOCK).

The UNLOCK instruction clears the Semaphore.
An UNLOCK instruction MUST quickly follow a LOCK instruction so that no CPU is blocked from accessing an element for too long.

Format
```
LOCK    semaphore   ;semaphore number 0..99
```

Example
```
;Semaphore 1 is used to protect data access
LOCK    1       ;if Semaphore 1 is Low (data is not being
CFB     H 100   ;accessed by another CPU), then call FB 100
```

Flags

ACCU	Set High or Low according to the state of the Semaphore
Status Flags	Unchanged

See also
[UNLOCK]

Practical example
A PCD4 is equipped with two CPUs. CPU 0 compares the contents of 2 registers while CPU 1 transfers BCD information into one of these two registers.
The use of semaphore 1 ensures that CPU 0 never compares the two registers while CPU 1 is executing the DIGI instructions, and is altering the contents of the registers.
If CPU 0 were to compare the registers at the same moment that CPU 1 was updating them, it might compare a new value with an old one.
Semaphore 1 also prevents CPU1 executing the DIGI instructions until CPU 0 has finished the CMP instruction.

```
CPU 0                CPU 1
...                  ...
LOCK   1             LOCK   1
CFB    H 10          CFB    H 100
...                  ...
```

```
FB      10            FB      100
CMP     R 88          DIGI    2
        R 89                  I 16
UNLOCK 1                      R 88
EFB                   DIGI    2
                              I 24
                              R 89
                      UNLOCK 1
                      EFB
```

12.7 NOP - No Operation

Description
Do-nothing instruction.
Used for patching out other instructions, for leaving space in the code for future additions or modifications, or for introducing very short delays.

Format
```
NOP    ;has no operand
```

Example
```
NOP    ;does nothing

NOP    ;space for 3 instructions, or short delay (depends on CPU speed)
NOP
NOP
```

Flags
Unchanged.

Practical example
An exciting program which, despite its complexity, does absolutely nothing at all.

```
    COB     0
            0
    NOP
    ECOB
```

12.8 OUTS - Set Element from ACCU Slow

Description
This instruction is not supported by new PCD types (NT systems, PCD3, PCD2.M480 etc), please use OUT.

The specified element, usually an Output, is set to the state of the ACCU.
This is the same as the OUT instruction, except that the timing on the PCD I/O bus is slightly slower, and it is therefore suitable for slow I/O modules.
Use this instruction to access Analogue modules PCA2.W2x/W3x.
The program execution speed is not affected.

Format
```
OUTS[X]  [=] element (i)   ;I O F
```

Example

```
OUTS    O 32
```

Flags

Unchanged.

See also

OUT

STHS

Practical example

The analogue value of channel 0 from a PCA2.W2x (base address 96) must be read and stored in Register 100.
After the conversion is made with the OUTS instruction, 8 binary bits can be read starting from the module base address + 8 (=104)

```
    COB     0
            0
    ...
    ACC     H         ;ensure the ACCU is High so OUTS executed
    OUTS    O 96      ;select analog channel
    ...               ;delay about 100 ms *)
    CPB     RD_VAL    ;read the analogue value
    ...
    ECOB

    PB      RD_VAL
    BITIR   8         ;read 8 bits binary in reversed order
            I 104     ;from I/O address 104..111
            R 100     ;into Register 100
    EPB
```

*) The analogue module PCA2.W2x has a conversion time of <= 100 ms. This wait function can be done by inserting a number of consecutive NOP instructions.
The number of NOPs depends on the CPU's processor speed.

12.9 PID - PID Control Algorithm

Description

Implements a PID algorithm using data defined in an array of 13 Registers.

Register	Meaning	Symbol		
+0	New Result	Yn	*	size is 'm' bits
+1	Previous Result	Yn 1	*	
+2	New Controlled Variable	Xn	w	size is 'm' bits
+3	Prev. Controlled Variable	Xn 1	*	
+4	Reference Variable	Wn	w	size is 'm' bits
+5	Prev. Set point Variable	Wn 1	*	
+6	Proportional Factor	Fp	w	* 256
+7	Integral Factor	Fi	w	* 256
+8	Derivative Factor	Fd	w	* 256
+9	Dead Range	Dr	w	

+10	Cold Start Y	Ys	w	Starting value for Yn
+11	Precision in bits	m	w	m = 8, 12 or 16 bits
+12	Workspace	Zs	*	

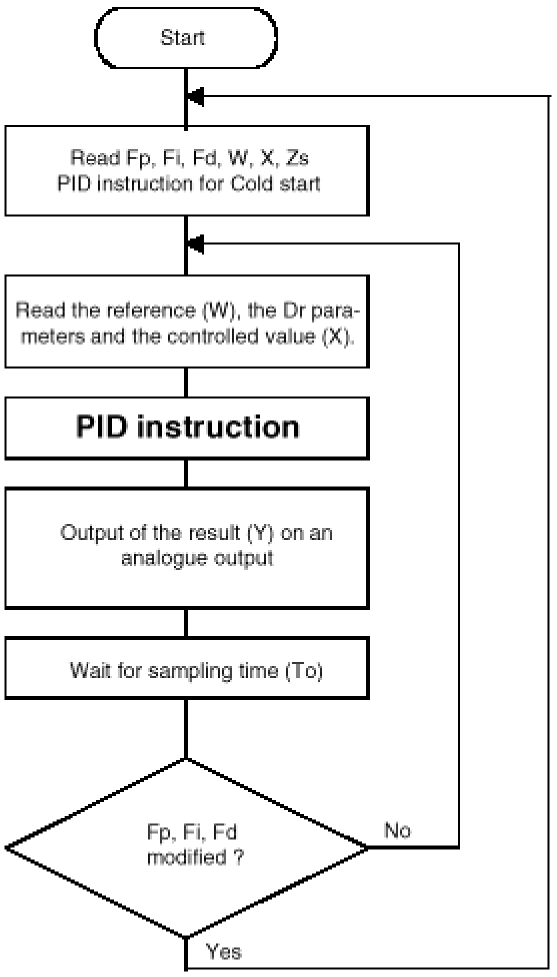
* These values are handled by the PID instruction.
w These values must be written into the register by the user program.

Format
```
PID    [=] reg    ;reg is the lowest address of 13 Registers
```

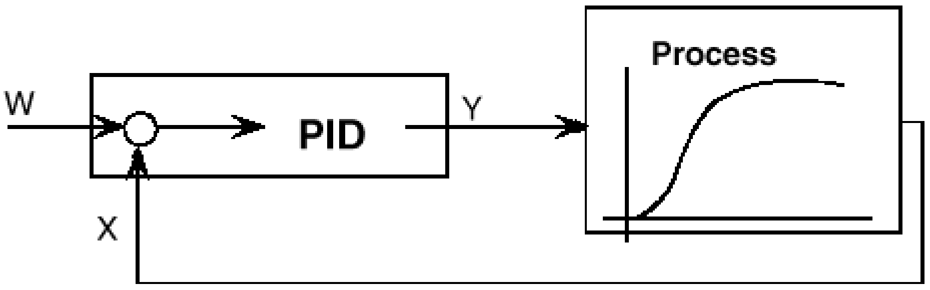
Example
```
PID    R 1000     ;use R 1000..1012 for the PID control data
```

Flags
Unchanged.

Practical example
Flowchart of typical PID control loop:


Start
Read Fp, Fi, Fd, W, X, Zs
PID instruction for Cold start
Read the reference (W), the Dr para-meters and the controlled value (X).
PID instruction
Output of the result (Y) on an analogue output
Wait for sampling time (To)
Fp, Fi, Fd modified ?
No
Yes

PID Instruction Details



New Result Yn:
This is the actual result to control the process determined by the system program from the following equation with Zs = Zs + (Wn - Xn):

$$Y_n = \frac{F_p}{256} * \left\{ \left(W_n - X_n\right) + \frac{F_i}{256} * Z_s + \frac{F_d}{256} * \left[\left(W_n - W_{n-1}\right) - \left(X_n - X_{n-1}\right)\right] \right\}$$

If the result exceeds the declared precision in bits, it will be limited to its maximum value (m bits) or, in case of a negative result, it will be set to 0.

Previous Result Yn1
This is the old result determined in the previous operation.

Controlled Variable Xn
The controlled variable Xn is read from the process and written to the register (R+2) by the user program.
The controlled variable should be maximum 'm' bits

Previous Controlled Variable Xn1
This is the old controlled variable used in the previous arithmetic operation.

Reference Wn
The reference (setpoint) is written to the register (R+4) by the user program. The reference should be maximum 'm' bits.

Previous Reference Wn1
This is the old reference used in the previous arithmetic operation.

Proportional Factor FP
This factor determines the proportional (amplification) characteristic of the regulator and is written to the register (R+6) by the user program.
When calculating, only the 16 lower bits are used (0..65535)
The Proportional factor is determined as follows:

$$F_p = \frac{1}{X_p} * 256$$

where Xp = Proportional band
Note: To enter a proportional band of 5%, the Fp factor must be set to:
(1 / 0.05) * 256 = 5120

A cold start of the PID must be executed after a modification of Fp or Fi

Integral Factor Fi
This factor determines the integral characteristic of the regulator and is written to the register (R+7) by

the user program.
When calculating, only the 16 lower bits are used (0..65535)
The Integral factor is determined as follows:

Fi = (To / Ti) * 256

where
To : sampling time of the PID instruction
Ti : integral time
A cold start of the PID must be executed after a modification of Fp or Fi

Derivative Factor Fd
This factor determines the derivative characteristic of the regulator and is written to the register (R+8) by the user program.
When calculating, only the 16 lower bits are used (0..65535)
The Derivative factor is determined as follows:

$$F_d = \frac{T_d}{T_0} * 256$$

where
To : sampling time of the PID instruction
Td : derivative time

Dead Range Dr
The dead range defines the range in which the variations of the controlled variable may occur without causing a modification of the Result variable (Yn).

Cold Start YS
This value is used as starting value for Yn by the system program.
As soon as the user program writes a value other than 0 to the cold start register, a cold start calculation is made: Yn = Ys

Yn1	=	Ys
Zs	=	[(Ys * 256/Fp) (Wn Xn)] *256/Fi
Wn1	=	Wn
Xn1	=	Xn

The value of Ys is automatically reset to 0 by the system program after being used once and will not be used again.

For a Cold Start with an output value of 0, the Ys register must be set to -1.

When Fi = 0, the Yn value can not be initialised with a Cold Start. A Cold Start is however recommended to initialise the workspace register.
In this case, the Ys value is ignored, the Zs register is set to 0 and Yn takes the value of the proportional part of the algorithm.

Note: Changing from manual to automatic control is a typical application of a cold start calculation. In order to achieve a smooth transition, Ys may be set equal to the currently output variable (Yn).

Resolution m
The maximum values of X, W, Yn and Ys are determined by the resolution.
If m = 8: 8 bits are used (0..255)
If m = 12: 12 bits are used (0..4095)
If m = 16: 16 bits are used (0..65535)

The resolution is mostly defined by the analog module used for the Result variable output.
If the resolution for the input and output are not the same, the Yn value must be adapted after the PID instruction.

Sampling Time
The sampling time To must be done outside the PID instruction with a timer.
In practice: To » 0,1 time constant of the process (To must be at least 80 ms)

Calculation capacity
The workspace register Zs has a maximum capacity of 231.
When using 16 bits values (m = 16), an overflow can occur; in this case the PID will not work properly.
To avoid this problem, the factor Fp must be ³ 2 if m = 16 (There is no problem when m = 8 or 12).

12.10 RDP - Read Peripheral

Description
Reads from a digital or analogue input.
This instruction is used by the Device Configurator's IO Handling feature.

Choose the mnemonic according to the size of the data to be read:

RDP	DWORD (32 bits), the default
RDPI	As RDP but the peripheral_adds is in a Register
RDPB	BYTE (8 bits), LS byte of source, other bytes set to 0
RDPBI	As RDPB but the peripheral_adds is in a Register
RDPW	WORD (16 bits), LS word of source, MS byte set to 0
RDPWI	As RDPW but the peripheral_adds is in a Register

Format
```
RDP   [K] peripheral_adds  ;0..65535, K is optional but ignored
       R  destination_reg  ;destination register

RDPI   R  register         ;peripheral_adds is in the Register
       R  destination_reg  ;destination register
```

Examples
```
RDP    8       ;read 32 bits from peripheral address 8
       R 45    ;into Register 45

RDPW   16      ;read 16 bits from peripheral address 16
       R 12    ;into Register 12 bits 15..0, bits 31..16 are set to zero

RDPBI  R 100   ;read 8 bits from the peripheral address in R 100
       R 101   ;into bits 7..0 of R 101, bits 15..8 are set to 0
```

See also
I/O Handling
WRP

12.11 RTIME - Read Time

Description
Reads the contents of the internal hardware clock into two Registers. The first Register is specified in the instruction. After the RTIME instruction, the Registers are set as follows:

Digit	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reg	0	0	0	0	Hour	Hour	Min	Min	Sec	Sec
Reg+1	0	Week	Week	WDay	Year	Year	Mon	Mon	Day	Day

Week	Week of year	1..53
WDay	Day of week	1..7 (Monday = 1, Sunday = 7)
Year	Year	0..99
Mon	Month of year	1..12
Day	Day of month	1..28/29/30/31 (month dependent)
Hour	Hour	0..23
Min	Minute	0..59
Sec	Second	0..59

The Register data is stored in binary, not in BCD, but can be moved or output in BCD using the DIGO instruction.

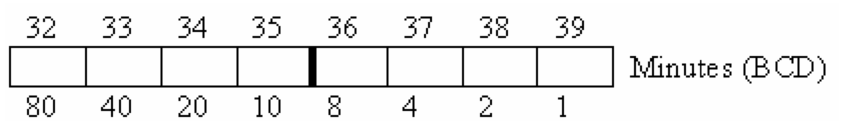
Format
```
RTIME    [=] reg    ;Register number R
```

Example
```
RTIME    R 10       ;clock is copied into Registers 10 and 11
```

Flags
Unchanged.

See also
WTIME
DIGO

Practical example
After switching on Input 3, the actual minutes of the clock should be displayed in binary BCD format on Outputs 32..39 :



```
    COB     0
            0
    STH     I 3
    DYN     F 3
    CPB     H 25
    ...
    ECOB

    PB      25
    RTIME   R 20
    MOV     R 20
            D 2
            R 99
            D 0
    MOV     R 20
            D 3
```

```
               R 99
               D 1
       DIGOR   2
               R 99
               O 32
       EPB
```

12.12 STHS - Start High Slow

Description

This instruction is not supported by new PCD types (NT systems, PCD3, PCD2.M480 etc), please use STH.

The ACCU is set to the logical state of the addressed element, usually an Input.
This is the same as the STH instruction, except that the timing on the PCD I/O bus is slightly slower, and it is therefore suitable for slow I/O modules.
Use this instruction to access Analogue modules PCA2.W2x/W3x.
Program execution speed is not significantly affected.

Format

```
STHS[X]  [=] element (i)   ;I O F
```

Example

```
STHS     I 25
```

Flags

The ACCU is set to the logical state of the specified element.

See also

OUTS
STH
Bit instructions

12.13 SYSCMP - System Compare

Description

The SYSCMP instruction is able to transform any Register into a pseudo Timer with a resolution of 1 millisecond.
It compares the sum of the first and second operands to the System Counter and sets the ACCU according to the result.
The System Counter is incremented every millisecond.

If the result of the addition is greater than the System Counter, the ACCU is set High (1).
If the result of the addition is smaller than or equal to the System Counter, then the ACCU is set Low (0).

The advantage of this instruction coupled to the instruction SYSRD K 7000 is that it is now possible to have Timers with a resolution of 1ms. It can also measure the time between two events to a resolution of 1ms.

To use it, first read the current System Counter value into a Register, add the number of milliseconds for the timeout, then compare the Register with the System Counter using SYSCMP.

Format

```
SYSCMP    reg     ;Register R
```

```
          value   ;value to compare, K or R
```

Example

```
SYSCMP    R 100   ;compare contents of Register 100 + 1500
          K 1500  ;to System Counter and set ACCU

SYSCMP    R 100   ;compare content of R100 + R101
          R 101   ;to System Counter and set ACCU
```

Flags

The ACCU is set according to the result.

See also

SYSRD

Practical example

Programming a high resolution Timer (1ms) with SYSRD and SYSCMP.

```
    COB     0
            0
    ...
    LD      R 100     ;load the time to wait in ms (1500)
            K 1500    ;into R 100
    SYSRD   K 7000    ;read the System Counter in R 101
            R 101
Wait:
    SYSCMP  R 101     ;compare System Counter to R100 + R101
            R 100     ;and set ACCU accordingly
    JR      H Wait    ;if ACCU = High (1) then loop
    ...
    ECOB
```

12.14 SYSRD - System Read

Description

Reads PCD system parameters like: PCD Device type, Firmware version, User program name, S-Bus parameters etc.
The data is transferred into Registers. Function codes, which are K constants or values in a Register, define which parameters will be read.

Tip: All new system functions are implemented using CSF Call System Function, see the new System Function libraries for details, using Project Manager's "Library Manager".

Format

```
SYSRD   func_code    ;function code, K or R
        result       ;Register for the result, or first of several Registers
```

The `func_code` can be a K constant, or a value in a Register.

Example

```
SYSRD    K 5000      ;read the PCD model in ASCII
         R 20        ;and put the result in R 20
```

Flags

If the function code does not exist, then XOB 13 is called and the Error flag is set.

See also
SYSWR
CSF

Read nonvolatile register (user EEPROM)
Some PCD models contain nonvolatile registers in EEPROM (usually programmed in the factory), which can be read into normal Registers.

Function Codes	2000. 2000+(n-1)	For EEPROM registers 0..n-1 where n is the number of EEPROM registers to be read: n = 5 for PCD1.M1x0, else n = 50
Result	Destination Register contains the data read from EEPROM	

Read FlashCard Status
Reads the FlashCard status and stores it in the ACCU and a Register.

Function Codes	3000	For PCD with FlashCard
	3100	For PCD with Onboard Flash (NT systems)
Result	ACCU set Low if OK ACCU set High if the Flash is busy and SYSWR was not executed, e.g. erasing or storing. Error Flag set is there's an error, see status bits. Destination Register contains the status bits, see below	

Note
Function code 9000 is now replaced by 3000, however the old function code (9000) is always supported on the system :
PCD2.M170 and PCD4.M170:
Bugs fixed : > #15 or later
New functionality :> $17 or later
PCD2.M480:
Bugs fixed : > #13 or later
New functionality : > $15 or later
For other systems:
Only the function code 3000 is supported, and the function code 9000 is reserved for LEDs.

Status bits

Bit	Description	Cause
0	No flash	No flash is fitted
1	No header config	There is no header/user program copied on the FlashCard
2	SYSWR not enabled	The DB/Text mode is not enable (memory allocation)
3	DB/Text does not exist	Incorrect DB/Text number
4	DB/Text size is not equal	DB/Text size is different, has been changed
5	Restored	DB/Text on FlashCard were restored because an error was detected
6	Buffer full	To much DB/Text are saved, memory is full
7	Already started	Last SYSWR 900x command was not finished before a new one was started
*8	Flash error	No backup DB is configured on the flash or on the SRAM. Impossible to access the flash, the bit is updated during the

		“initializing backup” or the “copying DB to flash”.
*9	Flash busy	Another job is working on flash.
*10	DB size error	DB size is zero. The bit is updated during the backup and restore DB.
11..15	Not used	
*16	Header different	The “User Program Backup Size” of the Flash is different from SRAM. The bit is updated during the “initializing backup” process.
*17	No flash card	Flash card is not present the bit is updated during the “initializing backup” process.
*18	No flash free	The “User Program Backup Size” >= ‘User flash size’. The bit is updated during the “initializing backup” process.
19..31	(for future use)	

* not used by PCD2.M170 and PCD4.M170

Read device type

Function Codes	5000	Read device type, ASCII format
	5010	Read device type, decimal format
Result	ASCII or decimal value as shown in the table below.	

Model	ASCII	Decimal
PCD1	`'  D1'`	1
PCD2	`'  D2'`	2
PCD3	`'<0><0>D3'`	3
PCD4	`'  D4'`	4
PCD6	`'  D6'`	6
PCS1	`'  S1'`	101

Read PCD type

Function Codes	5100	Read PCD type, ASCII format
	5110	Read PCD type, decimal format
Result	ASCII or decimal value. For ASCII, the first 3 digits of the subtype are returned. e.g. for PCD3.M5540 the result is `' M55'` For decimal, the first two digit of the subtype are returned. e.g. for PCD3.M5540 the result is 55	

Read FW version

Function Codes	5200	Read FW version, ASCII format
	5210	Read FW version, decimal format
Result	ASCII or hex value as shown in the table below	

Internal or experimental version numbers will be returned as 0FFFFFFFFh (-1 decimal).

	ASCII	Hex	Decimal
Internal version $39	‘ $39’	`0FFFFFFFFh`	-1
Official version V080	‘V080’	`000000080h`	128
Official version 1.14.31	'1.14'	`000011431h`	7075

Read CPU number
(Only for PCD4 and PCD6, others have only CPU 0)

Function Codes	5300	Read CPU number, ASCII format
	5310	Read CPU number, decimal format
Result	ASCII or decimal value For ASCII format, `'   0'.. '   6'` For decimal format, 0..6	

Read program name
Read the user program name into 2 Registers R+0 and R+1. Only the first 8 characters of the program name are read.

Function Code	5400	Read program name
Result	R+0	Upper 4 characters of program name in ASCII
	R+1	Lower 4 characters or program name in ASCII

Example
User program name is ‘MYPROG01’.
After execution of SYSRD :
R+0 = `'MYPR'`
R+1 = `'OG01'`

Read S-Bus Slave station number

Function Code	6000	Read S-Bus slave station number
Result	Station number 1..254, or -1 if S-Bus not configured	

Read S-Bus PGU TN delay

Function Code	6010	Read S-Bus PGU TN delay
Result	Delay in ms, or -1 is S-Bus not configured	

Read S-Bus PGU TS delay

Function Code	6020	Read S-Bus PGU TS delay
Result	Delay in ms, or -1 is S-Bus not configured	

Read S-Bus PGU timeout

Function Code	6030	Read S-Bus PGU timeout
Result	Delay in ms, or -1 is S-Bus not configured	

Read S-Bus PGU baudrate

Function Code	6040	Read S-Bus PGU baudrate
Result	Baud rate, or -1 is S-Bus not configured	

Read S-Bus PGU mode

Function Code	6050	Read S-Bus PGU mode
Result	Mode, see table below, or -1 is S-Bus not configured	

Mode	**Value**
BREAK without modems	0
PARITY without modems	1
DATA without modems	2
BREAK with modems	10
PARITY with modems	11
DATA with modems	12
S-BUS not configured	-1

Read S-Bus PGU port number

Function Code	6060	Read S-Bus PGU port number
Result	Port number, or -1 is S-Bus not configured	

Read current S-Bus PGU level

Function Code	6070	Read S-Bus PGU level
Result	1 = Level 1 S-Bus (Reduced Protocol) 2 = Level 2 S-Bus (Full Protocol) -1 = S-Bus not configured	

Read current PGU owner (S-Bus or P8 protocol)
(Obsolete, old PCD4 and PCD6 only)

Function Code	6080	Read current PGU port owner
Result	CPU number, 0..6	

Read modem status byte
Reads the current status of the modem connection.
This information tells the user at what stage the modem is at in the initialization procedure.

Function Code	6100	Read modem status byte
Result	1	Init Port
	2	PCD waiting for modem connection
	3...39	PCD initialising the modem.
	40	Reassign serial port for mode.

	45..49	Connection to modem has been lost. This is an intermediate status before the modem in reinitialized.
	50	Everything is OK and modem is ready to connect
	51...90	Connect/Dial
	100	Connected

Read modem initialization string

Read the specified modem string from the user program extended header into the block of registers starting with base address Ry.

Function Codes	6500	Read modem type string
	6510	Read modem reset string
	6520	Read modem initialization string
Result	String is read into Register block, see example below	

Example

```
;The modem initialization string is stored in the extended header
;in the PCD: "AT&Q0S0=2\r"
;Predefined text greater than length of modem string, contains spaces (20h)
TEXT 100 "                    "
...
SYSRD   K 6520
        R 1000       ;read string into register block
PUT     R 1000
        X 100        ;move string to TEXT 100
...
;Text after execution contains string followed by spaces
TEXT 100 "AT&Q0S0=2\r          "
```

Note: The space character is ignored by modems so the space characters stored at the end of the modem string will have no affect.

Read port mode

Read the mode of port 0..6. The mode is returned in a register in ASCII format.
If the port doesn't exist (no channel or the module Fxxx is not plugged in), the Error flag is set and the Register value is 0.

Function Codes	6600	Read Port 0 mode
	6601	Read Port 1 mode
	6606	Read Port 6 mode
Result	Mode is returned in the lower 3 bytes of the Register, the MS byte is 0.	

Mode	**Configured**	**Assigned (SASI)**
P8	PG0	
PGU with S-BUS	PG1	
S-BUS PGU	SP0 SP1 SP2 *	
MODEM (data mode)	MC2 (not connected) (not ready)	MC2 (not connected) (not ready)
	SP2 (connected) (ready)	SP2 (connected) (ready)
GATEWAY	GW0 GW1 GW2 *	GM0 GM1 GM2 *

MC		MC0..MC5
MD / SD		MD0 SD0
SM		SM0 SM1 SM2 *
SS		SS0 SS1 SS2 *
GS		GS0 GS1 GS2 *
MM		MM4

* 0 = break, 1 = parity, 2 = data

Example

```
SYSRD    6600    ;read channel 0 mode
         R 0     ;R 0 will contain "PG1" for PGU mode with S-Bus protocol
```

Read system counter

The PCD has a 32-bit internal counter which is incremented every millisecond. It is reset to zero only on power-up, it is not affected by a Restart.
The period of the counter is: **24 days 20 hours 31 min 23 sec 647 ms**, after this period the counter rolls over to 0.
The SYSCMP instruction can also be used to access the system counter, this instruction also works if a roll-over occurs.

Function Code	7000	Read system counter
Result	Any value between 0 and 2'147'483'647 decimal	

Read real time clock, Local time or UTC time

For Local time use function codes 70xx, for UTC time use 71xx. (The Local time zone is defined by the Device Configurator "Options -> Time Zone Code".)
Note: UTC time requires FW version 1.20.12 or later.
Date/time values can be read separately according to the function code.
The return value is always in decimal.

Function Code	7x50	Second
	7x51	Minute
	7x52	Hour
	7x53	Minute and second
	7x54	Hour and minute
	7x55	Hour, minute and second
	7x56	Hour, minute, second, millisecond
	7x57	Minute, second, millisecond
	7x58	Second, millisecond
	7x59	Millisecond
	7x60	Day
	7x61	Month
	7x62	Year
	7x63	Month and day
	7x64	Year and month
	7x65	Year, month and day
	7x70	Day of week
	7x71	Week of year
	7x72	Week of year and day of week
	7x81	Time and date (in two Registers, time in 1st Register)
	7x82	Time with ms and date (in two Registers, time in 1st Register)
	7x90 *	Seconds elapsed since midnight (00:00:00) 01/01/1970

x = 0 for Local time, x = 1 for UTC time (FW version 1.20.12 or later)
* With FW version 1.20.xx, `SYSRD 7090` returns the number of seconds elapsed as UTC time, not as Local time.

Examples
```
SYSRD    7055    ;read hour, minute and second, local time
         R 0     ;into Register 0
```
Result: R 0 = 120203

```
SYSRD    7181    ;read time and date, UTC time
         R 0     ;into Registers 0 and 1
```
Result: R 0 = 120203,   R 1 = 991130

Read DIL switch (PCD1 only)
Read the DIL switch of the PCD1.RIO or the Push Button on the PC1.M1x5 (PCD1 redesign).
This instruction is present but serves no purpose on the PCD1.M1_ .

Function Code	8000	Read DIL switch
Result	PCD1.RIO	The result is an 8-bit value with the 3 MS bytes set to zero. The bits 0 - 6 are reserved for the station number (0..127) and bit 7 for the communications speed (38.4 Kbaud or 76.8 Kbaud).
	PCD1.M1x5	Bit 0 indicates the Push Button activation (1=released, 0=pushed). Bit 1..7 = 0
	PCD1.M1x0	Bit 0..7 = 0

12.15 SYSWR - System Write

Description
Allows modification of system information or initialization of system functions from the user program.

Tip: All new system functions are implemented using CSF Call System Function, see the new System Function libraries for details, using Project Manager's "Library Manager".

Format
```
SYSWR    func_code    ;function code, K or R
         value        ;value to write, K or R
```

The `func_code` and `value` can be K constants, or values in a Register.

Example
```
SYSWR    K 4014    ;initialize XOB 14 with a frequency
         K 10      ;of 10 ms
```

Flags
If the function code does not exist, the Error flag is set and XOB 13 is called if it exists.

See also
SYSRD
CSF

Select PID algorithm

Selects between the new and old PID algorithms. By default the new PID algorithm is active.

Function Code	998	Select PID algorithm
Value	0	New PID algorithm (the default)
	1	Old PID algorithm

NOTE

The old PID algorithm is not included in NT systems, therefore function code 998 is not supported by NT systems which use only the new PCD algorithm.

System Watchdog (not on PCD6 and PCD4 except M170)

Activates and triggers the "watchdog". If not triggered every 200mS, a restart cold is done, preceded by an optional call to XOB 0.

Function Code	1000	System watchdog
Value	0	Deactivate WDOG
	1	Activate WDOG and make a restart cold if not retriggered within 200mS
	2	Activate WDOG and call XOB 0 before making a restart cold if not retriggered within 200mS.

Once the watchdog is activated the instruction must be repeated continually within 200mS intervals.
A watchdog XOB 0 is distinguished from the power down XOB 0 from the initial error massage written into the History List.
When the WDOG is activated the message “XOB0 WDOG START” is written into the History List, for a power down XOB 0 the message is “XOB 0 START EXEC”.

Write nonvolatile register (user EEPROM)

Some PCD systems are fitted with nonvolatile registers in EEPROM, which can be written with values from normal Registers.

Function Codes	2000..2000+(n-1)	For nonvolatile registers 0..n-1 n = 50, or 5 for PCD1.M1x0
Value	any	K or source Register containing the value to be written to EEPROM

WARNING

A maximum of 100,000 user writes are permitted to the EEPROM so do not execute this instruction frequently (cyclically) in your program.
This SYSWR instruction takes 20mS to execute so cannot be used in XOB 0.

Copy to/from FlashCard

Copies Text/DB memory to/from RAM memory from/to the FlashCard. This is only for PCD types which have a FlashCard or onboard Flash.
Note: Before using this function, please use SYSRD 3000 (Read Flash Status) to determine if the Flash memory is busy.

Function Codes	3000	Copy Text/DB from RAM to FlashCard
	3001	Restore Text/DB memory from FlashCard to RAM
	3002	Clear FlashCard, all Texts/DBs on the FlashCard are deleted)
	3100	Copy Text/DB from RAM to onboard Flash (NT systems)
	3101	Restore Text/DB from onboard Flash to RAM (NT systems)

	3102	Clear internal Flash, all saved Texts/DBs are deleted (NT systems)

Note
Function code 9000 is now replaced by 3000, however the old function code 9000 is still supported on these systems :

PCD2.M170 and PCD4.M170:

Bugs fixed	: > #15 or later
New functionality	: > $17 or later

PCD2.M480:

Bugs fixed	: > #13 or later
New functionality	: > $15 or later

For other systems:

Only the function code 3000 is supported, and the function code 9000 is reserved for LEDs.

WARNING
This SYSWR instruction takes a long time to execute so cannot be used in XOB 0.

Set XOB overflow limit

XOBs 14/15/17/18/19/20/25 all work using a queuing mechanism.
If an XOB is active then the pending XOB is placed in a queue which has a maximum size of 127 entries per XOB.
If this limit is surpassed then XOB 7 is called and the queue is cleared. The error message 'SYSTEM OVERLOAD' is written into the History List.
This limit of 127 entries can sometimes be too large for real time applications so it is now possible to define a user limit with this instruction.
This limit is common to all XOBs which can be queued.

Function Code	4000	Set XOB overflow limit
Value	0..127	New queue size

Enable/disable new XOB state change

Enable or disable the new state change mechanism for the XOB1/2 which is only called on state change. NT systems only.
ACCU = 0 on change error, ACCU = 1 no error.

Function Code	4001	Enable/disable new XOB mechanism
Value	0	Disable new XOB mechanism
	1	Enable new XOB mechanism

Enable/disable XOB 5/13

Enable or disable XOBs 5 or 13.
In some cases the execution of these XOBs after they have been called can complicate the execution of the user program. For this reason it is now possible to disable these XOBs.
If the XOB is disabled but not programmed, the error LED will not be set.

Function Codes	4005	XOB 5
	4013	XOB 13
Values	0	Disable the XOB
	1	Enable the XOB
	2	Clear the Error Flag in the current COB and in the active XOB For XOB 13 only (4013)

Install XOB 14 /15 (25 to 29 for NT systems, PCD3 and PCD2.M480)
Configure periodic XOB with the frequency defined in Ky or Ry.
It is possible to configure two periodic XOBs with a frequency from 5ms up to 1000s in 1mS steps.
The value in Ky or Ry is given in ms, if it's zero then the XOB is deactivated.
This instruction can be executed at any time.
If an XOB is already being executed when an XOB becomes pending then it will be queued until a time when there is no XOB active and it can be executed.
The XOBs are only executed if the CPU is in Run or Conditional Run.

Function Codes	4014	Configure XOB 14
	4015	Configure XOB 15
	4025..4029	Configure XOBs 25..29 (NY systems only)
Value	5..1'000'000	Time in milliseconds (min. is 1 for NT systems)

Execute XOB 17 /18 / 19
Execute the XOB specified in Rx or Kx on the CPU specified in Ky or Ry.
The XOBs 17/18/19 are user XOBs which can be invoked using S-Bus telegrams or the user program.
The XOBs are only executed if the CPU is in Run or Conditional Run.

Function Codes	4017	Execute XOB 17
	4018	Execute XOB 18
	4019	Execute XOB 19
Values	0..6	CPU number on which XOB will be invoked (PCD4 or PCD6 only)
	7	Call XOB on this CPU
	8	Call XOB on all CPUs

Write S-Bus station number
Changes the S-Bus station number to the value in Ky or Ry.
The S-Bus station number is changed even if the user program is in write-protected RAM, Flash or EPROM.

Function Code	6000	Write S-Bus station number
Value	0..254	New S-Bus station number

WARNING
A maximum of 100,000 user writes are permitted to the EEPROM so do not execute this instruction frequently (cyclically) in your program.
The SYSWR instruction takes 20ms to execute so it should not be used in XOB 0.

Convert FFP to/from IEEE
Converts between FFP (Motorola Fast Floating Point format) and IEEE floating point format.
Once a value is converted to IEEE format then no FFP floating point operations (FADD etc) can be carried out on the value.

Note
New PCDs now fully support IEEE values, see Floating Point Instructions.

Function Codes	7000	FFP to IEEE format
	7001	IEEE to FFP format
Value	R	Register containing the floating point value.

		The result is stored in the same Register. Only Registers are allowed, not K constants.

Write real time clock, Local time or UTC time

For Local time use function codes 70xx, for UTC time use 71xx. (The Local time zone is defined by the Device Configurator "Options -> Time Zone Code".)
Note: UTC time requires FW version 1.20.12 or later.
Date/time values can be written separately according to the function code.
Each value uses 2 digits, for example: 12h 2mins and 3 sec is written as 120203.

Function Codes	7x50	Second
	7x51	Minute
	7x52	Hour
	7x53	Minute and second
	7x54	Hour and minute
	7x55	Hour, minute and second
	7x56	Hour, minute, second, millisecond
	7x57	Minute, second, millisecond
	7x58	Second, millisecond
	7x59	Millisecond
	7x60	Day
	7x61	Month
	7x62	Year (< 100)
	7x63	Month and day
	7x64	Year and month
	7x65	Year, month and day
	7x81	Time and date (in two Registers, time in 1st Register)
	7x82	Time with ms and date (in two Registers, time in 1st Register)
Value	R	BCD value which depends in Function Code For example: 12h 2m 3s = 120203 Only Registers are allowed, not K constants.

x = 0 for Local time, x = 1 for UTC time (FW version 1.20.12 or later)

Examples

```
LD      R 0
        120203    ;12:02.03
SYSWR   7055      ;write hour, minute and second, local time
        R 0

LD      R 0
        120203    ;12:02.03
LD      R 1
        991130    ;30/11/99
SYSWR   7181      ;write time and date, UTC time
        R 0
```

Control reset push-button on the PCD1 redesign

Function Code	8000	Control reset push-button
Value	0	Deactivate push-button interrupt detection
	1	Activate Push button interrupt detection with restart cold /halt at push detection (default)

Set LED color for PCS1.C8xx and PCD3.Mxxx

PCS1.C8
Controls the color of the LED on the PCS1.C8. The LED is always switched off when the PCD is in Stop.
When the PCD runs, the LED will be turned on with the last used color.

Function Code	9000	Control LED colour
Value	0	Yellow
	1	Red
	2	Green
	3	Turn off

Notes
When the user program is stopped, the LED is switched off. It will be turned on with the same colour when the program runs again.
This command only works in Run, it does not work in step-by-step mode.

PCD3.Mxxx
By default the LED is used as the Error LED.
Using SYSWR 9000 reconfigures it as the user LED, which is independent of the system state (run/stop/error/halt).
After a restart it is re-initialized as the Error LED.

Function Code	9000	Turn on/off LED
Value	0	Turn OFF
	1	Turn ON

Flash Copy SYSWR 9000

Function code 9000 is now replaced by 3000, however the old function code (9000) is still supported on these systems :
PCD2M170 and PCD4M170:

Bug fix:	> #15 or newer
New functionality:	> $17 or newer

PCD2M480:

Bug fix:	> #13 or newer
New functionality:	> $15 or newer

Other systems:
Only the function code 3000 is supported, and the function code 9000 is reserved for LEDs.

Backup RAM or RAM+RTC for Compaktregler (PCS1.C8xx)

Configures how the backup capacitor is used. 20 days backup for RAM only, or 5 days backup for both RTS and RAM.

Function Code	9001	Backup RAM or RAM+RTC
Value	0	RAM only (20 days)
	1	RAM and RTC (5 days)

Enable serial port 0 (PCS1.C8xx)

Only for PCS1C8xx, FW 0A0 or later.

On the PCS1, port 0 can be used for the both the modem and the PGU connection.
Until FW version 0A0 the PGU port could only be used in PGU mode, but not in S-Bus PGU mode.
This was improved by introducing a new SYSWR instruction:

Function Code	9002	Enable/disable serial port 0
Value	0	Default setting, port 0 is used for the internal modem, the D-SUB connector cannot be used
		Port 0 uses the D-SUB connector, the internal modem cannot be used

Examples

```
SYSWR    K 9002     ;use port 0 for internal modem
         K 0        ;the D-SUB connector is disabled

SYSWR    K 9002     ;use port 0 for the D-SUB connector
         K 1        ;the internal modem can't be used
```

The user program can execute these instructions at any time to switch between the modem and the D-Sub connector.
This means that in some cases communication is lost when executing the instruction, for instance when online with S-Bus on the D-Sub port and switching to the modem.
Note that once the port is switched, it will stay in this mode also after a restart. The setting is stored in battery-backed RAM so the state will be switched to default when the supercap is discharged or the battery is replaced.

Tip: The port can also be controlled using the Online Debugger.
Put the PCD into Stop, and enter the SYSWR instruction using S-Bug's 'Instruction' command:
To switch port 0 to the internal modem:
```
> Instruction SYSWR K 9002, K 0 <Enter>
```
To switch port 0 to the D-Sub connector:
```
> Instruction SYSWR K 9002, K 1 <Enter>
```

IMPORTANT
The DSR signal (pin 6) of the D-Sub connector is used to recognize the PGU cable, indicating that the PG5 is connected.
As soon as the DSR signal is detected as High, port 0 is assigned as the PGU port and the previous configuration will be disabled.

12.16 TEST - Test Hardware

Description
Conditionally or unconditionally tests selected hardware of the PCD.

If any test fails, the test is aborted, and the ACCU is set Low (0).
If all the selected tests pass, the ACCU is set High (1).

Individual tests are selected as follows:

Value	Bit Number	Test Decription
	11	
400	10	Public Memory Loss
200	9	Extension Memory Corruption
100	8	System RAM Memory
	7	
40	6	System Firmware Checksum
20	5	Serial Ports
10	4	Real Time Clock
	3	
4	2	User Program Checksum
2	1	User Program RAM
1	0	Public RAM (F, T, C, R, mailbox)

For every bit set, the corresponding test is done.
Tests 0 and 5 are executed if the tested CPU is the only one in Run; if any other CPUs are running, these tests are NOT performed.

Note
Some of the tests are very slow, and should not be done during normal operation of the PCD, run the tests on startup or during an idling period.
The operand cannot be supplied as a Function Block parameter.

Format
```
TEST    [cc] value   ;cc = condition code, H L P Z N E
                     ;value defines the tests to run, see above
```

Example
```
TEST    50   ;test System Firmware Checksum (40) and Real Time Clock (10)

TEST    L 4  ;if ACCU = L (0), then verify checksum of the user program (4)
```

Flags
ACCU set High (1) if all tests pass, Low (0) if any test fails.

See also

Public RAM Test (Value = 1)
Tests the RAM which contains the F T C R media with a save-write-read-compare-restore operation.
This test is not performed if another CPU is in Run in a multiprocessor environment (PCD4 or PCD6).

ACCU	Error Flag	Description
0	1	Another CPU was in RUN
0	0	A Public RAM error was detected
1	X	Public RAM is OK

User Program RAM Test (Value = 2)
Tests the code and text RAM with a save-write-read-compare-restore operation.
If the memory is not RAM or if the RAM is write-protected, then the User Program Checksum test is

performed.

ACCU	Error Flag	Description
0	1	Program Header invalid
0	0	RAM is faulty
1	0	RAM is OK

User Program Checksum Test (Value = 4)
Calculates the checksum of the code and text memory and compares it with the checksum stored in the header.

ACCU	Error Flag	Description
0	1	Program Header invalid
0	0	Checksum is not OK
1	0	Checksum is OK

Real Time Clock Test (Value = 10)
Checks the existence of the RTC and tests if it is incrementing correctly.
Any other CPU accessing the RTC at the same time as this test is being performed will be blocked for up to 15ms.

ACCU	Error Flag	Description
0	1	RTC does not exist
0	0	RTC is faulty
1	0	RTC is OK

Serial Port Test (Value = 20)
Test the serial ports by initializing the port to local loopback mode and then transmitting a test pattern and verifying the reception of the same pattern.
The test is not performed if any of the serial port have been assigned with SASI.

ACCU	Error Flag	Description
0	1	One of the serial channel is assigned
0	0	Port is not OK
1	0	Serial channels are OK

System Firmware Checksum Test (Value = 40)
The firmware system EPROMs are checked.

ACCU	Error Flag	Description
0	0	Checksum is invalid
1	0	Checksum is OK

System RAM Memory Test (Value = 100)

The system RAM chips are checked.

ACCU	Error Flag	Description
0	0	Checksum is invalid
1	0	Checksum is OK

Extension Memory Corruption Test (Value = 200)
Test Extension Memory (RAM) which can be corrupted by backup battery or supercap discharging.

ACCU	Error Flag	Description
0	0	Memory extension was corrupted
1	0	No corruption has occurred

Public Memory Loss Test (Value = 400)
If the test pattern stored in the mailbox is not valid when it is tested during the startup routine then it is assumed that
the Public RAM has been corrupted during power down due to backup battery or supercap discharging.

ACCU	Error Flag	Description
0	0	Public Memory was corrupted
1	0	No corruption has occurred

12.17 UNLOCK - Unlock Semaphore

This instruction is only needed in the old PCD4 and PCD6 which could have more than one CPU that could share the same data.
It is not needed in PCDs with only one CPU.

Description
UNLOCK in conjunction with LOCK, is used to prevent access conflicts when several CPUs read or write the same elements.
100 Semaphores (special flags) are available (099).
The UNLOCK instruction clears the Semaphore.

Format
```
UNLOCK    semaphore   ;semaphore number 0..99
```

Example
```
UNLOCK    1   ;semaphore 1 is set Low
```

Flags

ACCU	Unchanged
Status Flags	Unchanged

See also
LOCK which has a practical example of semaphore use.

12.18 WRP - Write Peripheral

Description
Writes to a digital or analogue output.
This instruction is used by the Device Configurator's IO Handling feature.

Choose the mnemonic according to the size of the data to be written:

WRP	DWORD (32 bits), the default
WRPI	As WRP but the peripheral_adds is in a Register
WRPB	BYTE (8 bits), LS byte of source is written
WRPBI	As WRPB but the peripheral_adds is in a Register
WRPW	WORD (16 bits), LS word of source is written
WRPWI	As WRPW but the peripheral_adds is in a Register

Format
```
WRP  [K] peripheral_adds  ;0..65535, K is optional but ignored
      R  source_reg       ;source register

WRPI  R  register         ;peripheral_adds is in the Register
      R  source_reg       ;source register
```

Examples
```
WRP    32      ;write 32 bits to peripheral address 32
       R 10    ;from Register 10

WRPBI  R 16    ;write to the peripheral address in R 16
       R 17    ;bits 7..0 of Register 17
```

See also
I/O Handling
RDP

12.19 WTIME - Write Time

Description
Writes the contents of two Registers to the internal hardware clock.
The first of the two Registers is specified in the instruction.
The format of the Register contents is the same as for the RTIME instruction:
BCD values can be loaded into the Registers from Flags etc. using the DIGI instruction.

Format
```
WTIME    [=] reg  ;source Register R
```

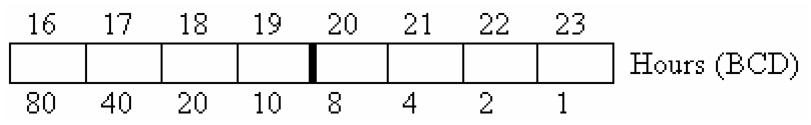
Example
```
WTIME    R 500    ;loads clock from Registers 500 and 501
```

Flags
Unchanged

See also
RTIME
DIGI

Practical example
After switching on Input 4, the hours of the clock should be set on a new value.

The new value is to be read from the BCD switches on Inputs 16..23.



```
;Inputs        16 17 18 19   20 21 22 23
;Hours (BCD)   8x 4x 2x 1x   x8 x4 x2 x1

COB      0
         0
STH      I 4
DYN      F 4
CPB      H 25
...
ECOB

PB       25
RTIME    R 200
DIGIR    2
         I 16
         R 199
MOV      R 199
         D 2
         R 200
         D 4
MOV      R 199
         D 1
         R 200
         D 5
WTIME    R 200
EPB
```

13 Media Pointer Instructions

A "media pointer" is the memory address of a Register, Flag, I/O, T/C, DB or Text in the PCD's memory. The media pointer is loaded into a Register so it can be used to load or store BYTE, WORD or DWORD values. An optional byte offset can be used, which is added to the media pointer address and allows access to the full range of media from a single base address.

For example, if the offset 4 is added to a register media address, it will point to the first byte of the next register.

Note: Media pointer instructions are available from firmware version 1.16.69.

XLA	Load Address
XLD[B|W]	Load Data (DWORD, BYTE, WORD)
XST[B|W]	Store Data (DWORD, BYTE, WORD)

If a BYTE or WORD value is loaded into a Register, you may want to sign-extend the value to a DWORD:

EXTB	Sign Extend BYTE
EXTW	Sign Extend WORD

The operator @MPTR( ) can also be used to obtain a 32-bit media pointer.

13.1 XLA - Load Address

Description
Loads the media pointer address of a Flag, I/O, T/C, Register, Text or DB into a Register.
An optional byte offset can also be added or subtracted.
This instruction is available from firmware version 1.16.69.

Format
```
XLA    R reg    ;register to receive the media pointer
       source   ;<media expression> [ ,|+|- <offset expression> ]
```

Examples
```
XLA    R 100    ;load the media pointer of F 32 into R 100
       F 32

Symbol1 EQU R     ;dynamic address
XLA    R 101       ;load media pointer to Symbol1 with offset 4 bytes
       Symbol1 + 4

XLA    R 102       ;load media pointer to Symbol1 with offset 4 bytes
       Symbol1, 4
```

Flags
Unchanged.

See also
XLD Load Data
XST Store Data
@MPTR() Get media pointer

Media Pointer Address Format
The media pointer address is a 32-bit value with bits as follows:

```
Normal media pointer for I|O|F|T|C|R (not TEXT or DB)

3322 2222 2222 1111 111  1100 0000 0000
1098 7654 3210 9876 5432 1098 7654 3210
0000 0ttt 0000 ssss ssss ssss ssss ssss
bit 31..27 = 00000
bit 26..24 = ttt = type, 3 bits
             0 = null pointer (all 0s)
             1 = Flag
             2 = Input/Output
             3 = Timer/Counter
             4 = Register
             5 = Task Register, current task (COB/XOB)
             6 = Task Flag, current task (COB/XOB)
             7 = unused
bit 23..20 = 0000
bit 19..0  = ssss .. ssss = byte offset, 20 bits

TEXT/DB media pointer

3322 2222 2222 1111 111  1100 0000 0000
1098 7654 3210 9876 5432 1098 7654 3210
10nn nnnn nnnn nnnn ssss ssss ssss ssss
bit 31..30 = 10
bit 29..16 = nn nnnn nnnn nnnn   = DB or TEXT number, 14 bits
bit 15..0  = ssss ssss ssss ssss = byte offset 0..65535, 16 bits
```

13.2 XLD - Load Data

Description
Loads the data pointed to by a media pointer with optional BYTE offset, into a Register.
There are separate instructions for loading a BYTE, WORD or DWORD value.
If a WORD or BYTE is loaded into a 32-bit Register, the upper 16 or 24 bits are set to 0.
You can sign-extend a signed BYTE or WORD to a signed DWORD using EXTB/EXTW.
This instruction is available from firmware version 1.16.69.

To load a Flag or I/O value, use `XLDB` to load the LS bit of a Register with the 1-bit value. Bits 31..1 are set to 0. `XLD` and `XLDW` will not work with Flags or I/Os (Error status flag set).

Format
```
XLD    R source    ;register containing the media pointer to be read
       R|K offset  ;BYTE offset from the source address (R or K)
       R reg       ;register to receive the 32-bit DWORD data value

XLDW   R source    ;register containing the media pointer to be read
       R|K offset  ;BYTE offset to the source address (R or K)
       R reg       ;register to receive the 16-bit WORD data value

XLDB   R source    ;register containing the media pointer to be read
       R|K offset  ;BYTE offset to the source address (R or K)
       R reg       ;register to receive the 8-bit BYTE data value
```

Example

```
XLA    R 0         ;R 0 := media pointer address of R 100
       R 1
LD     R 1         ;R 1 := data value 012345678h
       012345678h
;load byte 3 of R 100 into R 2
XLDB   R 0         ;media address in R 0, points to R 100
       3           ;byte offset (R 100 + 3 bytes)
       R 2         ;loads byte 3 of R 100 into R 2
;Result : R 2 := 078h
```

Note!
Offsets are in BYTEs, and can update an address to point to the next item. For example, an offset of 4 bytes added to a Register address will point to the next Register. In the example above, if the offset was 4 bytes, it would load the most significant byte of the NEXT Register R 101. Using offsets, the full range of media can be accessed from a single base pointer.

Flags

ACCU		Unchanged
Status Flags	E	Set High if XLD or XLDW is used with Flags or I/Os
		Other flags are unchanged.

See also
XLA Load Address
XST Store Data
EXTB/EXTW Sign Extension (convert signed BYTE or WORD to a signed DWORD)

13.3 XST - Store Data

Description
Stores the data pointed to by a media pointer plus optional BYTE offset into a Register.
There are separate instructions for storing a BYTE, WORD or DWORD value.
If a BYTE or WORD is stored into a 32-bit Register or DB item, the other bits are unchanged.
This instruction is available from firmware version 1.16.69.

Format

```
XST    R source    ;register containing the DWORD value to be stored
       R dest      ;register containing the destination media pointer
       R|K offset  ;BYTE offset to the destination address

XSTW   R source    ;register containing the WORD value to be stored
       R dest      ;register containing the destination media pointer
       R|K offset  ;BYTE offset to the destination address

XSTB   R source    ;register containing the BYTE value to be stored
       R dest      ;register containing the destination media pointer
       R|K offset  ;BYTE offset to the destination address
```

Example

```
XLA    R 10  ;load media pointer to R 12 into R 10
       R 12
XSTW   R 11  ;store WORD in R 11
       R 10  ;to media pointer (+ offset) defined in R 10
       K 10  ;offset is 2 - writes to bits 15..0 of R 12
```

Note!
Offsets are in BYTEs, and can update an address to point to the next item. For example, an offset of

4 bytes added to a Register address will point to the next Register. Using offsets, the full range of media can be accessed from a single base pointer.

Flags
Unchanged.

See also
XLA Load Address
XLD Load Data
EXTB/EXTW Sign Extension (converts signed BYTE or WORD to signed DWORD)

14 Declarations

These statements are for assigning values and comments to symbolic names.

`PUBL`	Public: Allows a symbol to be referenced from other files
`EXTN`	Extrenal: Allows a symbol declared in another file to be referenced
`EQU`	Equate: Permanently assigns a value and comment to a symbol name
`DEF`	Define: Temporarily assigns a value and comment to a symbol name
`LEQU and LDEF`	Declares an symbol which is local to a Macro
`GEQU and GDEF`	Declares a symbol in a Macro is can be accessed from outside the Macro.
`DOC`	Assigns a comment to an address or value which has no symbol
`TEQU`	Defines Temporary Data
`STR`	Strings

14.1 PUBL - Public

Description
Makes a symbol Public so that it can be accessed from any file by using EXTN (External).
If not Public, symbols can only be accessed from within the file which defines them.
The symbol should first be defined with an EQU statement.

Tip:
PUBL has been superseded by PEQU.
Instead of using two PUBL and EQU statements, public symbols can now be declared with a single PEQU statement.

Notes
- Labels cannot be declared public, but '$' may be used to generate a public label.
- Defined symbols (using DEF) cannot be made public.
- Forward references are allowed, the symbol can be defined after the PUBL statement.

Format
```
PUBL symbol [[,]symbol]... [;comment]
```

One or more symbolic names can follow the PUBL statement. Each symbol must be separated by one or more spaces or tabs, and/or by a comma. The comma is not required, but can be present if preferred.

Examples
```
Symbol1 EQU R     ;this is a public symbol
PUBL Symbol1

PUBL Symbol2, Symbol3, Symbol4
Symbol2 EQU R     ;so are these
Symbol3 EQU R
Symbol4 EQU F 66
```

See also
EXTN
PEQU
Scope of symbols

14.2 PEQU - Public Equate

Description
Declares a Public symbol which can be accessed from other files using EXTN (External).
For more details see EQU, it is the same.
This replaces the separate PUBL and EQU statements.

Format
```
symbol PEQU [type] [attribute] [value] [;comment]
```

Example
```
MySymbol PEQU R 45 ;this is a public symbol
```

See also
EQU
PUBL
EXTN
Scope of symbols

14.3 EXTN - External

Description
Declares an External symbol which is declared as Public in another file.
If a symbol is declared Public in one file (using PUBL), it can be referenced from another file by declaring it as External with EXTN.
It can also be assigned an optional type (I O F PB COB etc), which allows the Build to do type checking when the external symbol is used.

Tip: For some Macros and FBoxes it is important that the External's type is correct, because the type may be used to generate code for that particular data type.
For example, a Register and a Constant need different code to process them.

Notes
- The symbol's actual value is unknown until the files are linked together and the associated PUBL symbol is found.
- Forward references are allowed, the symbol can be defined after the EXTN statement.

Format
```
EXTN symbol [type] [[,] type]... [;comment]
```

One or more symbol names and optional types can follow the EXTN statement.
Each symbol must be separated by one or more spaces or tabs, and/or by a comma.
The comma is not required, but can be used if preferred.

Example
```
EXTN MySymbol F    ;this is an external symbol
EXTN Symbol1 R, Symbol2 F, Symbol3 DB
```

See also
PUBL
PEQU
Scope of symbols

14.4 EQU - Equate

Description

EQU declares a symbol and assigns the value of an expression to the symbol name.
An optional type can precede the expression to give the symbol a data type, see Typed Symbols.
If the optional expression contains an external, the resulting symbol also has external scope.
For R T C F and block types, the expression can be omitted, and the assembler will then automatically assign an address, this is known as dynamic address allocation.

Other details:

- The same symbol can be declared only once with EQU in the same file.
- If you want to declare the same symbol several times in the same file but with different values, you can use DEF.
- Symbols which are local to a block can be declared using LEQU inside the block.
- For symbols which are local to Macros, see LEQU and LDEF. For Macro symbols which can be accessed from outside a Macro, see GEQU and GDEF.
- If a comment is supplied, the comment is stored in the PCD file's symbol table.

Format

```
symbol EQU [type] [attribute] [expression] [:= init_value] [;comment]
```

Examples

```
MySymbol    EQU R 123  ;this is Register 123
Symbol0     EQU R      ;Register dynamic address assigned by linker
```

First-time initialization value

`:=` value defines an optional first-time initialisation value, which the downloader uses to initialize a Register, Counter or Flags when the program is first downloaded.

Arrays

Arrays of media can also be declared using square brackets `[..]`

```
    RegisterBase EQU R [10]
```

This declares an array of 10 Registers, with an dynamically assigned address for the base Register. Further symbols can then be declared as offsets from this symbol. The assembler prevents accesses to addresses outside the array.

```
    Register1  EQU  RegisterBase+0    ;same address as RegisterBase
    Register2  EQU  RegisterBase+1    ;next register
    Register9  EQU  RegisterBase+9    ;the last register
    Register10 EQU  RegisterBase+10   ;ERROR! Array bounds overflow
```

Volatile Flags and Text/DBs in RAM memory

For Flags, Texts and Data Blocks, an optional attribute can be given.
This is needed for the dynamic allocation of Flags in Volatile or Nonvolatile memory, or for Texts and Data Blocks in Text or Extension Memory.
To specify that a Flag is to be allocated from the volatile flags area, use the keyword `VOL` :

```
    VolatileFlag EQU F VOL      ;a flag in volatile memory
```

To specify that a Text or Data Block is to be allocated from the Extension Memory area (always RAM), use the keyword `RAM` :

```
    RamBasedText EQU TEXT RAM   ;a text in RAM extension memory
```

FLOAT and IEEE values (new in V2)

Registers can contain data as standard integers, or in floating point formats. Two floating point formats are supported:

Motorola Fast Floating Point (FFP) and the standard IEEE format. To indicate which format the Register contains, the FLOAT or IEEE attributes can be used:

```
    FFPRegister  EQU R FLOAT
    IEEERegister EQU R IEEE 100
```

This is used mainly by Fupla so that it does not mix integer and floating point data (incompatible formats).
At present, no checks are made by the assembler.

See also
DEF
LEQU and LDEF
Expression and Operators

14.5 DEF - Define

Description
Defines a symbol which can be re-defined with another type or value more than once in the same file. This is similar to EQU, except that the same symbol can be re-defined in the file with DEF, and the expression cannot contain an external or dynamic address.
DEF is used to define symbols whose values will change throughout the file, and is particularly useful in Macros which will be called several times in the same file.

Notes
- Forward references to DEFined symbols are not allowed, the symbol must be DEFined before it is used.
- DEF symbols cannot be made PUBLic. and their expressions cannot contain external symbols or symbols with dynamic addressing.
- DEF symbols appear in Project Manager's Data List view with the last value they were assigned when the file was assembled.
- DEF can also be used to define strings, see STR.

Format

```
symbol DEF [type] [attribute] expression  [;comment]
```

Example

```
DefSym DEF 0
...
DefSym DEF DefSym+1
...
DefSym DEF DefSym+1
```

See also
EQU
LEQU and LDEF
$FOR..$ENDFOR

14.6 LEQU, LDEF - Local Symbols

Description
These declare symbols which are local to the block (COB, FB etc) or to the Macro in which the statement appears.
They are the same as the EQU and DEF statements, except they are used inside macros or blocks.

This allows symbols to be defined within macros and blocks which do not produce "multi-defined symbol" errors if the macro is called more than once in the same file, or if temporary data uses the same symbol name in different blocks within the same file.

Symbols declared with LEQU or LDEF cannot be accessed directly by any nested macros (for this you should use GEQU or GDEF), and they cannot be accessed from outside the block.

Symbols declared with LEQU and LDEF cannot be made Public.

LEQU and LDEF symbols are not affected by $GROUP, the group name is not used.

Format

```
local_symbol_name LEQU [type] [expression]  [;comment]
local_symbol_name LDEF [type] [expression]  [;comment]
```

Example

```
BigMac      MACRO  param1
;Symbols local to a macro
Sym0        LEQU   R
Sym1        LEQU   R

            INC    param1

            ENDM


            COB    0
                   0
;Symbols local to the block
Reg0        LEQU   R
Reg1        LEQU   R

            bigmac(Reg0)

            ECOB

            PB     0
Reg0        LEQU   R
Reg1        LEQU   R

            bigmac(Reg0)

            EPB
```

See also

EQU
DEF
PEQU
GEQU and GDEF

Technical Info

To create a local symbol, the assembler adds a group name to the symbol to make it unique. A different group name is used for each block and each macro expansion. The prefix begins with an underscore, so you won't normally see these symbols in the "All Symbols" or "Data List" views in SPM unless you select "Internal Symbols".
Inside a macro, the group name is __mac__xxxxxx, inside a block the group name is __lequ__xxxxxx, where "xxxxxx" is a string which is unique to each macro call and each block.

You can see the group names in the Listing files. This is the code which is generated by the above example, taken from the listing file:

```
            COB    0
                   0
            NOP                    ;inserted by S-Asm for call to init code
;Symbols local to the block
__lequ__fghb89.Reg0        LEQU   R
__lequ__fghb89.Reg1        LEQU   R

            bigmac(__lequ__fghb89.Reg0)
;Symbols local to a macro
__mac__1h8phc0.Sym0        LEQU   R
__mac__1h8phc0.Sym1        LEQU   R
            INC    __lequ__fghb89.Reg0

            ECOB

            PB     0
__lequ__x4v9mx.Reg0        LEQU   R
__lequ__x4v9mx.Reg1        LEQU   R

            bigmac(__lequ__x4v9mx.Reg0)
;Symbols local to a macro
__mac__yrr75s.Sym0        LEQU   R
__mac__yrr75s.Sym1        LEQU   R
            INC    __lequ__x4v9mx.Reg0

            EPB
```

14.7 GEQU, GDEF - Global Macro Symbols

Description
These are the same as the EQU and DEF statements, except they are for use inside Macros, and define symbols which are local to the macro but can also be accessed by all other macros which are called from inside *this* macro (nested macro calls). This is often used inside FBoxes, which often use nested macros.

For local macro symbols use LEQU or LDEF.

GEQU and GDEF symbols are not affected by $GROUP, the group names are not used.

Format
```
global_symbol_name GEQU [type] [expression]  [;comment]
global_symbol_name GDEF [type] [expression]  [;comment]
```

Example

```
Mac1 MACRO
     LD   Reg1     ;accesses symbol Reg1 defined in macro Mac2
          100
     ENDM

Mac2 MACRO
Reg1 GEQU R 100    ;Reg1 declared
```

```
     Mac1( )       ;Reg1 can be accessed from this macro
     ENDM

     COB    0
            0
     Mac2( )
     ECOB
```

The above example generates this code:

```
     COB  0
          0
     NOP                           ;inserted by S-Asm for call to init code
     Mac2( )
__mac__g_1h8phc0.Reg1 GEQU R 100  ;Reg1 declared
     Mac1( )                       ;Reg1 can be accessed from this macro
     LD   __mac__g_1h8phc0.Reg1   ;accesses symbol Reg1 in macro Mac2
          100
     ECOB
```

14.8 DOC

Description
This is used to define a type/address without a symbol name, and to assign a comment to the type/ address without giving it a symbol name. The comment is used by the Documentation Generator, hence the name "DOC" meaning "documentation".

Data defined in the Symbol Editor without a symbol name is declared using DOC statements when it is saved in IL format.

Format
`DOC` *`type expression`* `[;comment]`

The *`type`* and *`expression`* (address) are both required.

Example
```
DOC I 16    ;On switch
```

Normally the data would also have a symbol name, so EQU (or PEQU) would be used as in this example:
```
OnSwitch EQU I 16 ;On switch
```

DOC is only needed if there's no symbol name.

14.9 TEQU - Temporary Data

Description
Defines a temporary Register or Flag for a block, which exists only while the current block runs. Use this for workspace data whose values do not need to be retained between block calls.

The data is always initialized to zero when the block starts. When the block ends, the data disappears.

Notes
- Even if the symbol is not used, temporary data is still allocated for it (unlike dynamic address

allocation, where an address is only allocated if it is used).
- Temporary data can be used only inside a block.
- Temporary data symbols cannot be made Public.
- Temporary data symbol names are not affected by $GROUP, the group names are not used.
- For each block, the assembler generates DEFTMP R and DEFTMP F instructions and inserts them at the end of the block.
  The DEFTMP instructions define the total number of temporary Registers or Flags used by the block.
- Temporary data cannot be used in $INIT, $COBSEG or $XOBSEG sections.
- Temporary data cannot be viewed in the Watch Window.
  It can only be viewed while stepping through a block using the Online Debugger (S-Bug), or the IL Editor (S-Edit).
- Temporary data can be written to using S-Bug, e.g. Write Register %1. See the descriptions below.
- If the PCD halts with a TEMPDATA ILLEGAL message, it is usually because of a missing DEFTMP M instruction in the COB or XOB.
  See DEFTMP for details.

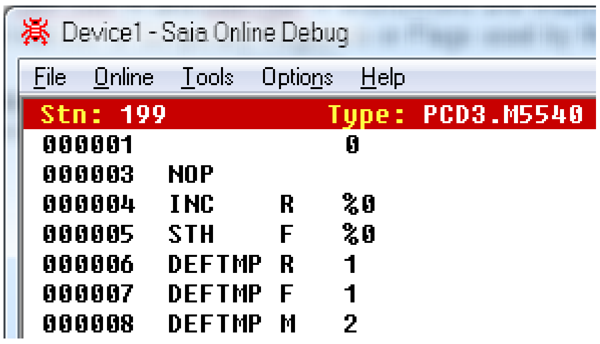
Format
```
symbol TEQU R|F [;comment]
```

Example
```
PB      0
;Declare temporary data, 2 Registers and 2 Flags
TempR1 TEQU R
TempR2 TEQU R
TempF1 TEQU F
TempF2 TEQU F
       ...        ;use the temp data
EPB
```

For the above example, the assembler generates these instructions and inserts them at the end of the block:
```
...
DEFTMP R 2   ;number of temporary Registers, inserted by S-Asm
DEFTMP F 2   ;number of temporary Flags, inserted by S-Asm
EPB
```

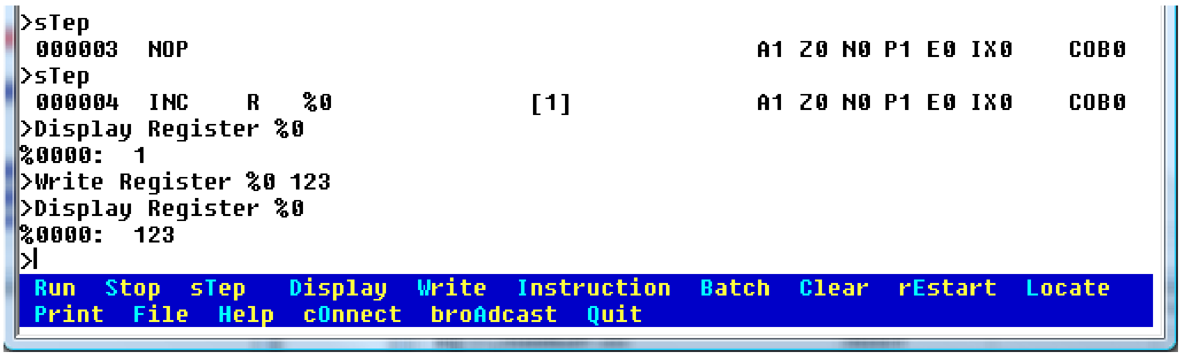
See also
DEFTMP

Using temporary data with the Online Debugger (S-Bug) and IL Editor (S-Edit)
Temporary addresses are indicated by a percent symbol % preceding the address.



```
Stn: 199             Type: PCD3.M5540
000001                0
000003  NOP
000004  INC     R     %0
000005  STH     F     %0
000006  DEFTMP  R     1
000007  DEFTMP  F     1
000008  DEFTMP  M     2
```

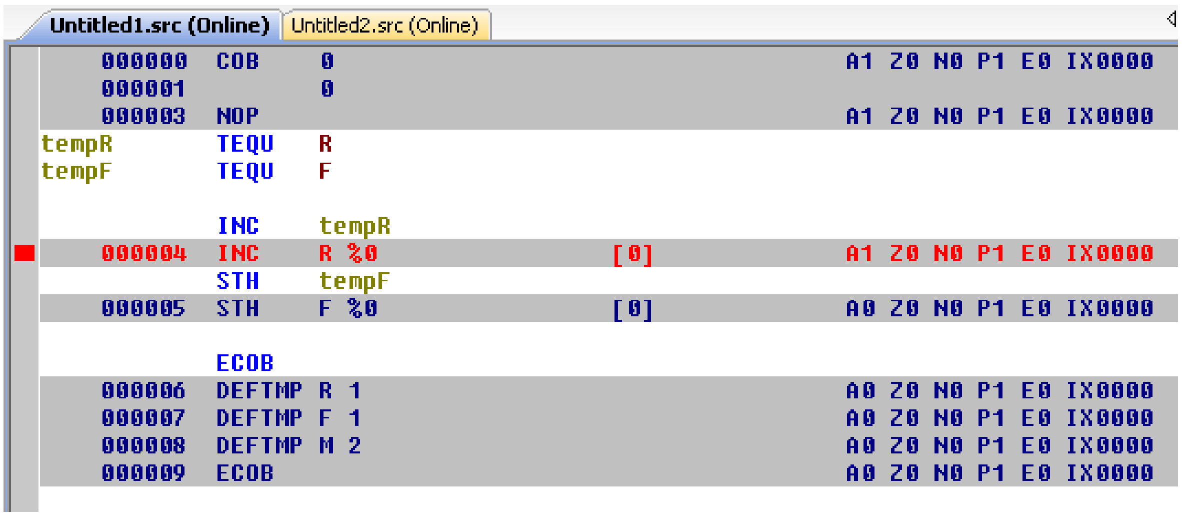
To display temporary data, precede the address on the command line with %.
To write temporary data, also use the % prefix for the address.



```
>sTep
 000003  NOP                                       A1 Z0 N0 P1 E0 IX0    COB0
>sTep
 000004  INC     R   %0            [1]             A1 Z0 N0 P1 E0 IX0    COB0
>Display Register %0
%0000:  1
>Write Register %0 123
>Display Register %0
%0000:  123
>|
```

Run Stop sTep Display Write Instruction Batch Clear rEstart Locate
Print File Help cOnnect broAdcast Quit

In the IL Editor (S-Edit), temporary data can be viewed when stepping through the program. You will also notice the DEFTMP instructions which are generated by the build.



```
        000000   COB     0                                A1 Z0 N0 P1 E0 IX0000
        000001           0
        000003   NOP                                      A1 Z0 N0 P1 E0 IX0000
tempR            TEQU    R
tempF            TEQU    F

                 INC     tempR
        000004   INC     R %0            [0]              A1 Z0 N0 P1 E0 IX0000
                 STH     tempF
        000005   STH     F %0            [0]              A0 Z0 N0 P1 E0 IX0000

                 ECOB
        000006   DEFTMP  R 1                              A0 Z0 N0 P1 E0 IX0000
        000007   DEFTMP  F 1                              A0 Z0 N0 P1 E0 IX0000
        000008   DEFTMP  M 2                              A0 Z0 N0 P1 E0 IX0000
        000009   ECOB                                     A0 Z0 N0 P1 E0 IX0000
```

15 Expressions

An expression is a combination of constants, symbols and operators, e.g. `Count + 1 - Index`, which is evaluated at build time. An operator starts with @ and is also evaluated at built time, e.g. `@POW(10, 2)`.

An expression or operator can appear in a declaration, a directive, or as an operand to an instruction, in fact, an expression can be used anywhere that a single number or symbol can be used. An expression can contains any combination of operators, symbol names and constants in any number base (decimal, hex, binary, floating point, IEEE), with the following exceptions:

- An expression can contain only one External symbol (see EXTN). If it contains an External reference then the result of the expression is also External.
- Operations permitted on an external symbol are:
  `external + constant_expression`
  `external - constant_expression`
  `constant_expression + external`
  All other operations are illegal on externals. A *`constant_expression`* is and expression which is fully evaluated (does not contain a reference to an external).
- If the expression contains typed symbols, the types must be the same, the expression evaluates to this type. This includes labels.
- All expressions are evaluated to 32-bit signed integers, overflow of expressions is not detected, e.g. 0FFFFFFFFH + 1 = 0.
- Expressions should not contain floating point values, since these are treated as 32-bit signed integers when the expression is evaluated, and the results will be wrong. However, the comparison operators "=" (is equal to) and "<>" (is not equal to) will work correctly with floating point numbers.
- An expression can contain forward references, but these should be used with care since the value of the symbol will be undefined on the first pass of the assembler. This could have disastrous effects, and may cause a "Pass 2 phase error".

Arithmetic Operators	+, - etc.
Bitwise Operators	For use with binary values
Comparison Operators	For comparing values
Operator Precedence	Order of evaluation
Special @Operators	Special operations which can be used in expressions

15.1 Arithmetic Integer Operators

`+`	(unary +ve, no operation since +ve is the default)
`-`	(unary -ve, 2's complement)
`+`	add
`-`	subtract
`*`	multiply
`/`	divide
`%`	modulo (returns the remainder)

15.2 Bitwise Binary Operators

`&`	AND
`|`	OR

^	XOR
!	NOT (1's complement)
>>	Shift right
<<	Shift left

AND, OR, XOR and NOT behave as unary operators on TRUE and FALSE values generated by the comparison operators below.

The shift operators are used as follows:

```
    constant_expr  <<  number_of_bits
```

The *number_of_bits* expression is evaluated, and the *constant_expression* is shifted left this number of bits.

15.3 Comparison Operators

=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

The result of expressions containing these is either TRUE (7FFFFFFFH) or FALSE (0).

These operators can appear only in expressions following conditional assembly directives.

Only = and <> can be used with FLOAT and IEEE values.

15.4 Operator Precedence

Operator precedence means the order in which operators are evaluated in an expression.
Operators with higher precedence are evaluated first.
Parentheses (....) can be used to change the order of precedence, operators in brackets are evaluated first, from left to right. Operators of equal precedence are also evaluated left to right.

1 highest	! + - (unary)
2	* / %
3	+ - (add/subtract)
4	<< >> (shift operators)
5	= <>
6	> >= < <=
7	& (binary AND)
8	^ (XOR)
9 lowest	| (OR)

16 $ Directives

All directives consist of a **$** immediately followed by the directive name. No space can appear between the **$** and the name.
Directives direct the assembler to do something special when the source module is assembled (built).
Directives are processed at assembly time, they are never executed by the PCD.

$AUTO	Defines address ranges for Dynamic Address Allocation.
$CHARSET	Selects the character set for all subsequent texts.
$COBSEG..$ENDCOBSEG	Places code in a COB.
$STATION	Defines the target S-BUS station number
$DNLDFILE	Declares a 'downloadable file', e.g. BACNet configuration
$ERROR	Displays an error message in the Message window.
$FATAL	Generates a fatal error, the build is stopped.
$GROUP	Declares symbol groups.
$IFxxxx..$ENDIF	Conditional assembly directives.
$IFEXIST	True if a file exists.
$INCLUDE	Includes another IL file.
$INIT, $ENDINIT	Places code in XOB 16, same as $XOBSEG 16.
$IPADDS	Define the target PCD's IP Address
$LIB	For importing library modules.
$LIST, $NOLIST, $EJECT	Controls output to listing files.
$NOXINIT, $ENDNOXINIT	Controls extension memory initialization.
$ONERROR	Defines text to be output after any error message.
$PCDVER	Defines target PCD types and F/W versions.
$REPORT	Displays text in the Message Window.
$SASI, $ENDSASI	For enclosing SASI instruction texts to turn on detailed error checking.
$SERIALNO	Defines the target PCD's serial number
$SKIP, $ENDSKIP	For patching out sections for code.
$TITLE, $STITLE	Defines the title and subtitles for the Listing files.
$USE, $IFUSED	Allow the use of an external symbol (defined in another module) in a conditional statement.
$WARNING	Displays a warning message in the Message Window.
$WRFILE	Writes formatted text to a file during the build.
$XOBSEG..$ENDXOBSEG	Places code in an XOB.
Using symbols in $directives	How to use symbols in $ERROR, $REPORT etc.

16.1 $ATTR

Description
This directive assigns an *attribute* *string* to the symbol whose definition follows $ATTR. The Symbol Editor uses $ATTR to assign the symbol Tags which are described below.

Format
```
$ATTR attribute_name=value
```

Tag names
$ATTR is most commonly used to define *tag names* for a symbol. A tag name is a text which indicates a property of the symbol, for example "HMI" might be used to indicate that the symbol is from the HMI Editor. Symbols can have more than one tag name. Tag names can be assigned from the Symbol Editor by clicking on the button in the symbol's Tags column, this generates the $ATTR directives when the symbol file is saved. There are new columns which show the tag names in the Symbol Editor and Project Manager's "Data List" view. The tag names can be used for sorting and filtering.

Attribute strings
$ATTR can also be used to define an attribute with a name and a string. This string can be referenced from IL code using the new @ATTR() special operator.

Format
```
$ATTR TAGS=tag1 [, tag2]...
Symbol EQU R 123   ;EQU PEQU LEQU symbol definition
```

Examples
If you configure IOs using the device Configurator, it generates symbols with the tag name `S_IO`:

```
;System symbols from IO definition
...
$ATTR TAGS=S_IO
IO.Slot0.DigitalInput0 EQU F 0  ;Digital input 0
$ATTR TAGS=S_IO
IO.Slot0.DigitalInput1 EQU F 1  ;Digital input 1
...
```

See also
@ATTR( ) - Returns an attribute's value
@STR( )

16.2 $CHARSET

Description
Selects the character set for all subsequent texts.

With the PG5, files are edited using the ANSI character set.
This directive causes S-Asm to convert the characters from ANSI to either the old DOS OEM character set, or to the GSM (Global System for Mobile communications, originally from Groupe Spécial Mobile) character set.

The default is ANSI (no conversion).

The character set is used from the $CHARSET directive to the end of the file.

This is mainly for use with terminals which do not support the ANSI character set.

For example, some Saia PCD terminals use the old DOS OEM character set, and some have selectable character sets.

The character set affects only the accented and special characters. The codes for the standard alphanumeric characters are the same in all character sets.

Note
In the GSM character set, the '@' character has the value 0, and can only be used in Texts 4000 and above.

Format
```
$CHARSET ANSI | OEM | GSM
```

Example
```
;ANSI text, the default
TEXT 0 "I'm ANSI"

;OEM text
$CHARSET OEM
TEXT 1 "I'm OEM"

;GSM text
$CHARSET GSM
TEXT 2 "I'm GSM"

;Back to ANSI
$CHARSET ANSI
```

In fact, all the texts in this example will be exactly the same, because the standard alphanumeric characters have the same codes.

16.3 $COBSEG .. $ENDCOBSEG, $XOBSEG .. $ENDXOBSEG

Description
These directives are similar to the existing $INIT..$ENDINIT directives which put code into XOB 16, except that code between $COBSEG..$ENDCOBSEG is put into a COB, and code between $XOBSEG..$ENDXOBSEG is put into an XOB. $XOBSEG 16 is the same as $INIT.

If the block is already defined in the user program, the code between these directives is inserted at the start of the block, before the code that is already defined.
If the block is not already defined in the program, it is created and added to the end of the user program.

These directives can be nested up to 10 deep, i.e. a $COBSEG can contain a $COBSEG which can contain a $XOBSEG or $INIT segment and so on to a depth of 10.

For $COBSEG, the `cob_number` is optional. If $COBSEG has no `cob_number` parameter, then the code between $COBSEG and $ENDCOBSEG is placed in an automatically allocated COB which is added to the end of the user program. This COB will be executed cyclically after all preceding COBs have been processed.
`cob_number` or `xob_number` define the number of the block into which the code will be inserted. This number must be an absolute value or symbol, it cannot be an external or dynamically allocated symbol.

The code is inserted by creating a PB containing all the code, and inserting a call to this PB at the start of the COB or XOB. Every COB and XOB has a NOP inserted at the start, and S-Asm replaces this with CPB to call $xxSEG code.

Format
```
$COBSEG [cob_number]
...
$ENDCOBSEG

$XOBSEG xob_number
...
$ENDXOBSEG
```

Example
```
    MyCob   EQU COB 10
    ...
    $COBSEG MyCob
    CFB     DoSomeStuff    ;code for COB MyCob
    $ENDCOBSEG

    $XOBSEG 16             ;same as $INIT
    LD      R 120          ;code for XOB 16
            0
    $ENDXOBSEG

    COB     MyCob
    LD      R 123
            456
    ECOB
```

This code is generated:
```
    COB     MyCob
    CPB     299            ;inserted by S-Asm
    LD      R 123
            456
    ECOB

    ;This is inserted by S-Asm
    PB      299
    CFB     DoSomeStuff
    EPB
```

See also
$INIT..$ENDINIT

16.4 $DNLDFILE

Description
Supported by PCDs with firmware version 1.14.0 and later.

Adds a 'downloadable file' to the list of additional files to be downloaded, e.g. a BACNet configuration file.
Downloadable files are downloaded into the PCD when the user program is downloaded.

Tip: The list of downloadable files for a PCD can be seen in the .map file which is found in Project Manager's 'Listing Files' branch.

Format

```
$DNLDFILE "[dir\]filename.ext",filetype[,number]
```

`dir\` is the optional drive and directory or relative path **on the PC**. If not present then the file is assumed to be in the device directory.
`filename.ext` is the name of the file, which must be a valid PCD file system file name: use the letters A..Z, numbers 0..9 or underscore '_'. Accented characters cannot be used. The PCD's file system always converts the file name to upper case.
`filetype` defines how the file will be processed by the PCD, and can be a value between 0 and 255 (00 and FF hex). Not all types are used, see the list below. For add-on tools, the file type is defined on the "Add-on Tools" dialog box in the "Downloadable Files" section.
`number` is optional, and is assumed to be zero if it is not supplied. Currently this is not used and can be left out.

These are the downloadable file types currently supported:

File Type	**Description**	**Directory in PCD ***
16 (10H)	BACNet configuration	-
17 (11H)	CANopen configuration	-
18 (12H)	LONIP configuration	-
32 (20H)	Configuration file, e.g. PCD.SCFG	PLC_SYS\CONFIG
33 (21H)	Program file, e.g. Ethernet RIO file	PLC_SYS\PCD_PROG
34 (22H)	Web page file	PLC_SYS\WEBPAGES

* The PCD's PLC_SYS directory is hidden, it cannot be accessed by Flash Explorer or other FTP browsers.

Example

This code is generated by Project Manager if the BACnet Configurator is used.
It causes the BACnet configuration file to be downloaded.

```
;Downloadable Files
$DNLDFILE "Untitled3.5bn",16
```

16.5 $ERROR

Description

Displays the user-defined error message in Project Manager's Messages window, and increments the error count.
This is often used inside a $IF..$ENDIF statement, which can be used to detect the error, i.e. if a user-defined constant was out of range.

The error message can contain expressions and macro parameter references, see Using symbols in $directives.

The @STR() and @ATTR() special operators can also be used, so you can display strings and symbol attributes.

Format

```
$ERROR error_message
```

Example

```
$IF Axis > 4
$ERROR "Axis" is out of range: @Axis@
$ENDIF
```

If Axis is greater than 4 (e.g. 10), the assembler generates an error message:
Error 165: Axis.src: Line 45: "Axis" is out of range: 10

See also
$FATAL
$WARNING
$ONERROR
$REPORT
Using symbols in $directives

16.6 $FATAL

Description
Generates a fatal error message, assembly is aborted, and no object file or listing is produced.
This is often used inside a $IF..$ENDIF statement, which can be used to detect the fatal error.
This directive might be used to ensure a particular symbol is correctly defined, or the correct version of an include file has been used.

The fatal error message can contain expressions and macro parameter references, see Using symbols in $directives.

The @STR() and @ATTR() special operators can also be used, so you can display strings and symbol attributes.

Format

```
$FATAL fatal_error_message
```

Example

```
$INCLUDE fred.inc
$IF FredsVersion < 3
$FATAL Wrong version of FRED.INC
$ENDIF
```

If symbol FredsVersion in include file FRED.INC is less than 3 then this fatal error is generated:
Fatal Error 20: TEST.SRC: Line 32: Wrong version of FRED.INC

See also
$ERROR
$WARNING
$ONERROR
$REPORT
Using symbols in $directives

16.7 $FBPARAM .. $ENDFBPARAM

Description
These directives are used to create a format definition of Function Block parameters.
They are used when creating a Function Block library, each library has a file ".lib" which defines the parameters for each FB in the library.

FBs with $FBPARAM definitions are shown in the IL Editor's "Function Selector" window, from where the FB call can be copied into the IL code.

The FB parameters are also validated by the build according to the $FBPARAM definition.

Each parameter is defined with a name, type, direction and comment, as described below. If the parameter is an array, the array size must also be given.

Format

```
$FBPARAM fb_name
    param_name type ['['array_size']'] direction [;comment]
    ...
$ENDFBPARAM [fb_name]
```

Where:

`fb_name`	The symbol name of the Function Block.
`param_name`	The symbol name of the parameter. This symbol may be used inside the FB code. It is not normally needed by the caller of the FB.
`type`	The data type of the parameter. In many cases more than one type is supported. This is indicated by an | (or) character between the types. These are the currently supported data types:

```
R
T
C
I
O
F
K
DB
TEXT
X
T|C
R|T|C
R|K
R|F
DB|X
DB|X|R
X|K
DB|K
DB|X|K
X|R
X|R|F
DB|R
X|R|K
DB|R|K
DB|X|R|K
F|K
I|O|F
ANY       Any type
BOOL      0 or 1
UINT      16-bit unsigned constant
INT       16-bit signed constant
RFLOAT    Register containing Motorola FFP value
```

`RIEEE`	`Register containing IEEE float value`

`array_size`	If the parameter is an array, its size must also be given. This is usually used only for Register and Flag types.
`direction`	Indicates if the parameter is read or written (or both) by the FB.

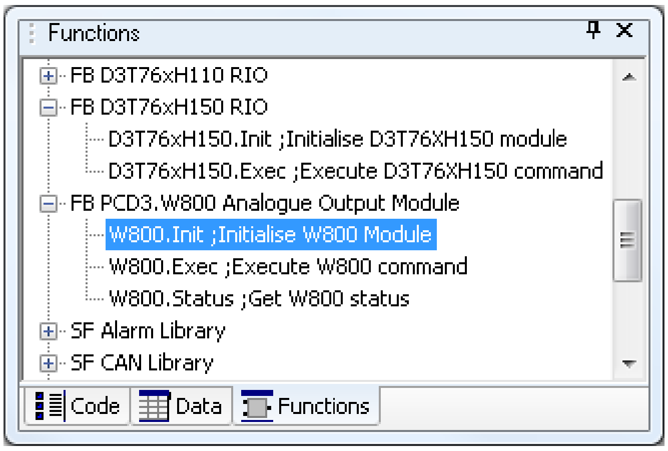
`IN`	Parameter is read, can be any data type.
`OUT`	Parameter is written, cannot be a constant type, e.g. K, INT, UINT, FLOAT, IEEE etc.
`INOUT`	Parameter is read and written, cannot be a constant type, e.g. K, INT, UINT, FLOAT, IEEE etc.

Examples

This example is from the PCD3.W800 FB Library, file D3W800_B_en.lib:

```
;PCD3.W800 FB LIBRARY

$include "D3W800_B.equ"

$FBPARAM W800.Init  ;Initialise W800 Module
ModuleNumber        K IN            ;=1  Module number (K 1..16)
CH0Scaling100       R IN            ;=2  CH0 100% scaling
CH0Scaling0         R IN            ;=3  CH0 0% scaling
CH0ResetValue       R IN            ;=4  CH0 Reset value
CH1Scaling100       R IN            ;=5  CH1 100% scaling
CH1Scaling0         R IN            ;=6  CH1 0% scaling
CH1ResetValue       R IN            ;=7  CH1 Reset value
CH2Scaling100       R IN            ;=8  CH2 100% scaling
CH2Scaling0         R IN            ;=9  CH2 0% scaling
CH2ResetValue       R IN            ;=10 CH2 Reset value
CH3Scaling100       R IN            ;=11 CH3 100% scaling
CH3Scaling0         R IN            ;=12 CH3 0% scaling
CH3ResetValue       R IN            ;=13 CH3 Reset value
InitFailed          F OUT           ;=14 Initialization failed
$ENDFBPARAM

$FBPARAM W800.Exec  ;Execute W800 command
ModuleNumber        K IN            ;=1 Module number (K 1..16)
Command             UINT IN         ;=2 Command
Value               R IN            ;=3 Value (R)
NotReady            F OUT           ;=4 Not ready flag
$ENDFBPARAM

$FBPARAM W800.Status ;Get W800 status
ModuleNumber        K IN            ;=1 Module number (K 1..16)
ManAuto0            F OUT           ;=2 MAN/AUTO0
ManAuto1            F OUT           ;=3 MAN/AUTO1
ManAuto2            F OUT           ;=4 MAN/AUTO2
CalError            F OUT           ;=5 Calibration error
W800Error           F OUT           ;=6 W800 error
NotReady            F OUT           ;=7 Not ready
$ENDFBPARAM
```

This is used by S-Edit to show the library in its "Function Selector" window:



FB D3T76xH110 RIO
FB D3T76xH150 RIO
D3T76xH150.Init ;Initialise D3T76XH150 module
D3T76xH150.Exec ;Execute D3T76XH150 command
FB PCD3.W800 Analogue Output Module
W800.Init ;Initialise W800 Module
W800.Exec ;Execute W800 command
W800.Status ;Get W800 status
SF Alarm Library
SF CAN Library

When the function is added to the IL code, a template is inserted which can be filled in with the actual parameters:

```
    CFB    W800.Init      ;Initialise W800 Module
                          ;=1 K IN, =1  Module number (K 1..16)
                          ;=2 R IN, =2  CH0 100% scaling
                          ;=3 R IN, =3  CH0 0% scaling
                          ;=4 R IN, =4  CH0 Reset value
                          ;=5 R IN, =5  CH1 100% scaling
                          ;=6 R IN, =6  CH1 0% scaling
                          ;=7 R IN, =7  CH1 Reset value
                          ;=8 R IN, =8  CH2 100% scaling
                          ;=9 R IN, =9  CH2 0% scaling
                          ;=10 R IN, =10 CH2 Reset value
                          ;=11 R IN, =11 CH3 100% scaling
                          ;=12 R IN, =12 CH3 0% scaling
                          ;=13 R IN, =13 CH3 Reset value
                          ;=14 F OUT, =14 Initialization failed
```

See also
CFB

16.8 $FOR .. $ENDFOR

From PG5 V2.1.300 ($2.1.261)

Description
It is often necessary to repeat a section of IL code several times. The IL code between `$FOR` and `$ENDFOR` is repeated the number of times specified by the *`start`* and *`end`* values. The value of *`symbol`* is initialized with the *`start`* value, and is incremented **by one** each time the code is repeated, up to and including the *`end`* value.

The loop control *`symbol`* is declared in the same way as it would be if using a DEF declaration. If you wanted to increment the value by something other than one, you can do this in the loop by re-DEFining the symbol as shown in the example below. But remember that it will still be incremented by 1 in the $FOR loop, so to increment by 2 you should add 1 using: `symbol DEF symbol + 1`.

You can also use other DEFined symbols inside the loop for more complex behaviour. DEFine the symbols outside the loop, and then re-DEFine them inside the loop. See the example below.

Format

```
$FOR symbol = start .. end
    ...             ;code to repeat
$ENDFOR
```

Examples

Clear Registers 100 to 109:

```
$FOR reg = 0 .. 9
    LD  R 100 + reg
        0
$ENDFOR
```

$FOR always increments the control symbol by 1, but if you wanted a decrementing value, or a value incremented by more than one, then you can use another DEFined symbol and modify its value inside the loop.
For example, to load Registers 100 to 109 with 10, 20, 30 … 90:

```
LoopSym10 DEF 10
$FOR LoopSym1 = 0 .. 9        ;LoopSym1 increments by 1
    LD  R 100 + LoopSym1
        LoopSym10
LoopSym10 DEF LoopSym10 + 10  ;LoopSym10 increments by 10
$ENDFOR
```

The loop can be exited at any time by using DEF to set the control *symbol* to a value beyond the *end* value.
If you are updating the control symbol to produce steps bigger than 1, remember that the $FOR loop also adds 1, so increment it by symbol - 1.

```
$FOR symbol = 0 .. 1000
$IF symbol > 500
symbol    DEF   1000        ;end the loop at 500
$ELSE
symbol    DEF   symbol + 9  ;symbol += 10 (9 + 1 for $FOR)
$ENDIF
$ENDFOR
```

Notes

- Macros cannot be called within $FOR .. $ENDFOR sections, but you can use $FOR inside macros.
- Nested $FOR statements are not allowed, but you can usually do the same thing with several $FOR statements, and/or with DEF symbol outside the loop.
- If there's an error in the $FOR statement, it will process the code just once, and give a "$ENDFOR without $FOR" error for the $ENDFOR statement.
- The start/end range can be anything, providing you update the symbol to restrict the loop count to the maximum of 65536.
- The maximum times a $FOR loop can be repeated is 65536, so you can't write infinite loops.
- You cannot generate more then 1'000'000 (1 million) lines of code in a single source file.

See also

DEFine

16.9 $GROUP

Description
All symbols defined between these directives are automatically prefixed with the group name. $GROUP directives can be nested up to 10 deep. Symbols defined within nested groups are prefixed by all the group names:

All symbols defined or referenced between the $GROUP..$ENDGROUP directives are assumed to be symbols from that group, unless prefixed with another group name or `"."`.
If the symbol has not been defined in that group then the assembler will search for a matching symbol outside the group.

The total length of group names and the symbol name cannot exceed 80 characters.

Notes
- Symbols with group names can also be defined like this, it's not necessary to use $GROUP:
  `Group0.Group1.Symbol EQU R`
- The $GROUP name is not used for LEQU and LDEF or GEQU and GDEF symbols because these symbols given prefixes by S-Asm to make them unique.
- Group names cannot be reserved words or IL mnemonics, e.g. TEST, but a sub-group name can be a reserved word if it is defined in this way:
  `$GROUP Main.Test`
- Top-level group names should be more than one character in length, because single-character group names are reserved for system symbols, e.g. `S.CPU.PcdType`.

Format
```
$GROUP group_name
  ...
$ENDGROUP
```

Group statements can be nested up to 10 deep:

```
$GROUP group_name
  $GROUP subgroup_name1
    $GROUP subgroup_name2
    ...
    $ENDGROUP
  $ENDGROUP
$ENDGROUP
```

Examples
```
$GROUP FRED                ;the group name is FRED
  SYMBOL1 EQU F 100        ;SYMBOL1 is in group FRED
  SYMBOL2 EQU F            ;so is SYMBOL2
$ENDGROUP
```

To reference a symbol in a group, either the group name can be used as a prefix to the symbol (separated by a full stop "."), or another $GROUP directive can be used to define the default group name. For example:

```
STH   FRED.SYMBOL1         ;SYMBOL1 in group FRED
$GROUP FRED                ;select group FRED
STH   SYMBOL1              ;references FRED.SYMBOL1
$ENDGROUP
```

When inside a $GROUP section, symbols outside the group can be referenced by using the "." prefix:

```
SYMBOL1 EQU I 0         ;not in a group

$GROUP FRED
SYMBOL1 EQU I 1         ;defines FRED.SYMBOL1
STH   SYMBOL1           ;references FRED.SYMBOL1
STH   .SYMBOL1          ;references SYMBOL1
$ENDGROUP
```

$GROUP directives can be nested up to 10 deep. Symbols defined within nested groups are prefixed by all the group names:

```
$GROUP FRED             ;start group FRED
SYMBOL1 EQU I 0         ;defines FRED.SYMBOL1

$GROUP TOM              ;start group FRED.TOM
SYMBOL1 EQU I 1         ;defines FRED.TOM.SYMBOL1
STH SYMBOL1             ;references FRED.TOM.SYMBOL1
$ENDGROUP               ;end group FRED.TOM

STH SYMBOL1             ;references FRED.SYMBOL1
$ENDGROUP               ;end group FRED
```

16.10 $IFxxx .. $ENDIF

Description

The conditional assembly directives allow code to be conditionally included or excluded at assembly time, according to the value or type of a symbol. They are evaluated when the program is assembled, not when it is executed. Each directive is terminated with $ENDIF, and may contain optional $ELSE or $ELSEIFxxx directives (xxx is the condition type, see below).

Conditional directives are typically used to produce different versions of a program where most of the code is the same for each version, depending on the application, PLC hardware or firmware version, or to include special code during the debug phase of a program. Some of directives are designed specifically for use inside MACROs , see $IFB, $IFNB and $IFE, $IFNE.

Statements following $IFxxx are processed if the condition is true, statements following $ELSE or $ELSIFxxx are processed if the condition is false.

Statements following $IFxxx, $ELSE or $ELSEIFxxx can be any statements or directives including further nested $IF statements.

Format

```
$IFxxxx conditional_expression
  statements
[ $ELSEIFxxxx conditional_expression
  statements ]
[ $ELSE
  statements ]
  $ENDIF
```

These are the $IFxxx directives:

$IF, $IFN - if, if not

`$IF` *`conditional_expression`*
If the result of *`conditional_expression`* is non-zero (true).
The *`conditional_expression`* is described below.

`$IFN` *`conditional_expression`*
If not: if the result of *`conditional_expression`* is zero (false).

For $IF and $IFN, *`conditional_expression`* uses a comparison operator:
`conditional_expression :=`
`constant_expression comparison_operator constant_expression`

For example:
```
$IF Symbol = 1
...
$ENDIF
```

Note: FLOAT and IEEE floating point values can only be used with = and <>, the other comparison operators **do not work with floating point values**.

$IFDEF, $IFNDEF - if defined, if not defined

`$IFDEF` *`symbol`*
`$IFNDEF` *`symbol`*
If the symbol name is/is not defined.

If the *`symbol`* is a STRING name, you must use `STR` so it looks in the string table
```
$IFDEF STR stringName
...
$ENDIF
```

$IFINC, $IFNINC - if $included, if not $included

`$IFINC`
`$IFNINC`
If this is/is not a $INCLUDE file (true if this file has been included in another file).
There is no *`conditional_expression`* for $IFINC and $IFNINC.

$IFTYPE - if type is

`$IFTYPE` *`expression = type`*
`$IFTYPE` *`expression <> type`*
If the expression's type is equal to type (R T C COB PB etc). expression will normally be just a symbol name.
Can also use <> in place of = for "not equal to" type, e.g.
`$IFTYPE Symbol <> R`.
If the symbol is external, then it must be declared with a type, e.g.
`EXTN Symbol R`
otherwise $IFTYPE will assume it is a constant (K).

For $IFTYPE, the *conditional_expression* is:

	`expr = type`	is equal to type, e.g. `Fred = R`, `R 0=R`
or	*`expr <> type`*	is NOT equal to type, e.g. `Fred<>R`

`type` can be:
`R T C I O F K = TEXT X DB DBX COB PB FB XOB IST ST TR I|O T|C IB STR`

```
$IFTYPE Symbol = R
$IFTYPE R 123 = R
$IFTYPE Symbol <> F
```

```
$IFTYPE myString = STR
```

$IFB, $IFNB - if blank, if not blank

$IFB	If blank
$IFNB	If not blank
$IFE	If equal
$IFNE	If not equal

These are for use inside macros for checking macro parameter strings, see $IFB, $IFNB, $IFE, $IFNE and Macros.
For $IFB and $IFNB, *conditional_expression* is any string or macro parameter name enclosed in angle brackets, see Macros.

$IFDYN, $IFNDYN - if dynamic address, if not dynamic address

$IFDYN *symbol*
$IFNDYN *symbol*
If symbol is/is not a dynamic address which is defined by the system.
Tip: Symbols with dynamic addresses should not be accessed by external systems using their addresses because the addresses can change. You can use $IFDYN to check these symbols and give a warning.

$IFEXIST - if file exists

$IFEXIST "*filename*"
True if the file exists, see $IFEXIST for details.

$IFLINKED - if the file is linked

$IFLINKED "*filename*"
True if the file is assembled and linked (is part of the program), see $IFLINKED for details.

$ELSEIF directives

These $ELSEIFxxxx directives are supported, they work in the same way as a combination of $ELSE with the associated $IF directive:

```
$ELSEIF      $ELSEIFN
$ELSEIFDEF   $ELSEIFNDEF
$ELSEIFB     $ELSEIFNB
$ELSEIFE     $ELSEIFNE
$ELSEIFINC   $ELSEIFNINC
$ELSEIFTYPE
```

Notes

- $IF statements can be nested up to 30 deep.
- The use of forward references in the *conditional_expression* and symbol is NOT allowed, except with $IFDEF and $IFNDEF.
  All symbols must be defined BEFORE they can be used in $IFxxx statements.
- $IFxxx cannot be used inside a multi-line instruction or FB parameter list etc, the entire instruction must be enclosed by the $IF..$ENDIF statements.
- IEEE and FLOAT floating point values can only be used with = and <>.

Examples

```
Version EQU 1           ;Define version number

$IF Version = 1
    STH I 1000          ;Code for version 1
$ELSEIF Version = 2
    STH I 10            ;Code for version 2
```

```
$ELSE
    STH F 100           ;Code for all other versions
$ENDIF

$IF version=1 & !testing ;If assembling version 1 and
    SET O 0             ;not testing the program
$ELSEIF !testing        ;Else if not testing
    SET O 32
$ENDIF

DEBUGGING EQU 1         ;Set DEBUGGING to 1 (true)

$IF DEBUGGING           ;If DEBUGGING is not zero (true)
label: JR label         ;then include this loop in the code
$ENDIF

$IFNINC                 ;If not $INCLUDEd file,
    FB ONE              ;then assemble the FB code
    OnSwitch DEF =1     ;FB parameter definitions, available
    Output DEF =2       ;to including module. FB code is only
    ...                 ;assembled once, it is not assembled
$ENDIF                  ;where it is $INCLUDEd.

$IFTYPE MySymbol = R       ;If MySymbol is a Register
    $REPORT MySymbol is a Register
$ELSEIFTYPE MySymbol <> X  ;If MySymbol is NOT a Text
    $REPORT MySymbol is not a text
$ENDIF
```

See also
$IFEXIST
$IFLINKED
$IFB, $IFNB
$IFE, $IFNE

16.11 $IFB, $IFNB

Description
These directives can be used to control the expansion of a macro according to the actual parameters supplied with the macro call.

$IFB and $IFNB mean "if blank" and "if not blank". These are used to determine if an actual parameter was supplied or not.

Notes
- Use $IFE and $IFNE to compare macro parameters.
- To compare symbol values use $IF or $IFN.
- To check symbol types use $IFTYPE.
- For detecting empty strings, if the string may contain '>', you can use `@STRLEN(param) = -1`. @STRLEN() is useful for detecting blank macro parameters when `$IFB <string>` fails if the string may contain the '>' character.

Format
```
$IFB <parameter>    ;The < and > must be present!!
...
```

```
$ELSE               ;and nested $IFB's are allowed
...
$ENDIF
```

Example 1
AND gate macro.

Macro definition:

```
ANDGATE MACRO INPUT1, INPUT2, OUTPUT
    STH     INPUT1
    ANH     INPUT2
    $IFNB <OUTPUT>      ;;if parameter "OUTPUT" is supplied (not blank)
    OUT     OUTPUT      ;;then write to OUTPUT, else
    $ENDIF              ;;leave result only in ACCU
ENDM
```

Macro call:

```
    ANDGATE(I 32, I 63)
```

Expands to:

```
    STH   I 32
    ANH   I 63
```

Example 2
`EXITM` ends the expansion of the macro, this can be used inside conditional directives.
This is useful to exit from deeply nested `$IFxxxx` statements. For example:

```
ANDGATE MACRO INPUT, OUTPUT
        STH   INPUT
        ANH   INPUT+1
        $IFB <OUTPUT>      ;If "OUTPUT" is blank,
        EXITM              ;stop macro expansion
        $ENDIF
        OUT OUTPUT         ;else write the output
ENDM
```

Macro call:	Expands to:
`ANDGATE (I 0)`	`STH   I 0`
	`ANH   I 1`
`ANDGATE (I 0, O 32)`	`STH   I 0`
	`ANH   I 1`
	`OUT   O 32`

See also
$IF
$IFE, $IFNE - If equal, if not equal
@STRLEN() Gets the length of a String

16.12 $IFE, $IFNE

Description
$IFE and $IFNE mean "if equal" and "if not equal". These compare an actual parameter string with a given string, or compare two actual parameter strings. The comparison is not case sensitive (unless certain accented characters are used).

Notes

- The strings themselves are compared, not the symbol's values.
- To compare symbol values and numbers use $IF or $IFN.
- To check symbol types use $IFTYPE.
- Do not use if the string may contain the '>' character.
- For detecting empty strings, if the string may contain '>' you can use `@STRLEN(param) = -1`.

Format

```
$IFE <param1> <param2>  ;Enclosing < > must be present!!
...                     ;$ELSE, and nested $IFB's allowed
$ENDIF
```

Example 1

Macro to load a value into a Register.
If the value is already in a Register we must use COPY, not LD.

```
LOADREG MACRO DestType, DesValue, SrcType, SrcValue
    ;;if SrcType string is "K" then use LD
    $IFE <SrcType> <K>
    LD     DestType DestValue
           SrcValue
    ;;if source type is not K then use COPY
    $ELSE
    COPY   SrcType SrcValue
           DestType DestValue
    $ENDIF
ENDM
```

Example 2

The above example can be done much more easily using the new $IFTYPE directive:

```
LOADREG MACRO Dest, Srce
    $IFTYPE <Srce> <K>
    LD     Dest
           Srce
    $ELSE
    COPY   Srce
           Dest
    $ENDIF
ENDM
```

See also

$IF
$IFB, $IFNB
$IFTYPE
@STRLEN() Gets the length of a String

16.13 $IFEXIST

Description

The result is true if the filename exists. The filename can be a full path, a relative path or just a file name. The filename can also be a macro parameter. If the file name contains spaces it must be enclosed in double quotes.

If the file name has no directory, the assembler searches for the file in these directories:
1. Device directory (of PG5 device)
2. Directory of main source file (if not same as device directory)

3. Library directories of libraries selected by the project's Library Manager

Format

```
$IFEXIST "filename.ext"
;this code is assembled only if "filename.ext" exists
;in the device directory or a selected library directory
...
$ENDIF
```

See also

$IF
$IFLINKED

16.14 $IFLINKED

Description

The result is true if the filename is assembled and linked with the user program.

The filename can be a full path, a relative path or just a file name, but only the file name (without path or extension) is checked, because it's not possible to linked more than pone file with the same file name.

The filename can also be a macro parameter.

If the file name contains spaces it must be enclosed in double quotes.

Format

```
$IFLINKED "filename.ext"
...
$ENDIF
```

See also

$IF
$IFEXIST

16.15 $INCLUDE

Description

When an include statement is encountered, the given file is read and processed as though it is part of the source file being assembled.

If the file name has no directory, the assembler searches for the file in these directories:

1. Device directory (of PG5 device)
2. Directory of main source file (if not same as device directory)
3. Library directories of libraries selected by the project's Library Manager

Include files can be nested up to 10 deep, this means that an included file can itself contain a $INCLUDE directive to include another file and so on until 9 further files have been included.

Tips:

- Avoid using full path names in $INCLUDE statements because it will cause a 'Can't open include file' error if the program is moved to another location. Instead, either use relative path names, or use only the file name and place the file in one of the library directories which are searched by S-Asm.
- In relative paths, '`..`' means the parent directory.

- You can also use directory tags instead of hard-wired directory names, see the example below.
- The listing file contains the full path of the file which was included, which is useful to verify that the correct file was used.

IMPORTANT
You cannot pass the include file name as a macro parameter, and $INCLUDE will not work within $IF statements inside a macro. This is because the file is included when the macro *definition* is processed, not when the macro is *called*.

Format
```
$INCLUDE "filename.ext"
```

The file name or path should be enclosed in double quotes `"..."`.

Examples
To include a file which is in the same directory as the file which includes it:
```
$include "Include.src"
```

This will include the file from the device's 'Project' directory which is used for Common Files:
```
$include "..\Include.src".
```

This will include MyFile.inc from two directories up:
```
$include "..\..\MyFile.inc"
```

File names can contain the directory tag <$LibsDir>, which is replaced by the path of the actual libraries directory:
```
$include "<$LibsDir>\Std\MyLib.inc"
```

See also
$IFINC, $IFNINC

16.16 $INIT .. $ENDINIT

Description
All code between these directives is placed in the start-up XOB 16, and executed on power up or restart cold.
This allows initialization code to be written close to the code which uses it, rather than in the XOB 16 module, and improves modularity.
The code is executed at the start of XOB 16, before any code in XOB 16 is processed.

If XOB 16 does not exist, it is created.
If XOB 16 is already defined, then the $INIT code is assembled into a PB, and the PB is called at the start of XOB 16.
This is done by converting the NOP at the start of the XOB into a CPB instruction.
The code is assembled in the order it appears in the source file, and the order in which the source files are linked.

Format
```
$INIT
...      ;code to be inserted at the start of XOB 16
$ENDINIT
```

Example

File 1 **File 2** **File 3** **After linkage**

```
PB 1           PB 2           XOB 16         XOB 16
$INIT          $INIT          LD R 0         CPB 299
LD R 1         LD R 2            0           LD   R 0
   100            200         EXOB              0
$ENDINIT       $ENDINIT                      EXOB
EPB            EPB
                                             PB 299
                                             LD R 1
                                                100
                                             LD R 2
                                                200
                                             EPB
```

S-Asm creates a PB using the last available PB number which has not been used.
All the $INIT code is inserted into this PB, and a call to the PB is inserted at the start of XOB 16.
If there is no $INIT code then this remains a NOP instruction.

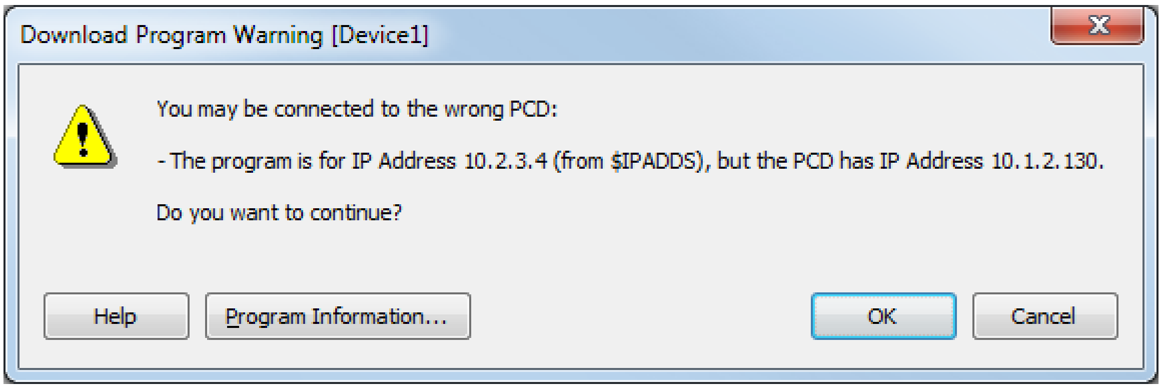
See also
$COBSEG, $XOBSEG

16.17 $IPADDS

Description
Specifies the IP address or IP address range of the PCD into which this program can be downloaded.
This directive can be used to ensure greater security on PCD networks.
A warning will be shown if an attempt is made to download the program into a PCD with the wrong IP address. For PCDs with two IP ports, this is the address of the first port.

Only one $IPADDS directive can appear in each PCD program.

Tip: You can also use the Downloader Option "Warn if different Station number or IP Address".


Download Program Warning [Device1]

You may be connected to the wrong PCD:

- The program is for IP Address 10.2.3.4 (from $IPADDS), but the PCD has IP Address 10.1.2.130.

Do you want to continue?

Format
```
$IPADDS firstAdds[,lastAdds]
```

Examples
```
;This program can be downloaded only to PCD with IP address 10.2.3.4
$IPADDS 10.2.3.4

;This program can be downloaded only to PCDs with IP address 10.2.3.1
;to 10.2.3.39
$IPADDS 10.2.3.1,10.2.3.39
```

See also
$SERIALNO
$STATION

16.18 $LIB

Description
Causes a library file to be assembled and/or linked with the program.

The code in one source file may require the presence of code from other source files or libraries, such as Program Blocks, Function Blocks, Texts or DBs which are referenced by code in the source file. To avoid having to manually add the name of all source files to be linked to the Project Tree list, the $LIB directive can be used to do this automatically.

If the file extension is '.obj' or '.obl' then it is assumed to be an object file and it is linked instead of assembled.
If filename has any other extension it is assumed to be a source file, and is assembled (if required) before linking the resulting '.obj' file.

$LIB tells the assembler and linker that the filename must also be assembled and/or linked.
It is only assembled and linked once, even if there are several $LIBs for the same file. It is not the same as $INCLUDE, where the file is read and assembled every time.

Usually an include file will also be needed which defines the public symbols in the $LIB file because the main source file does not know anything about the contents of the $LIB file, therefore the most logical place to put the $LIB directive is in this include file. This is often done for libraries.

If the file does not exist in the device directory then the selected library directories are searched in the same was as for $include files.

Relative paths and directory tags can also be used, see the examples for $include.

When linking, the list of $LIB file names is shown after the list of object files being linked.

Any number of $LIB files can be defined. $LIB can also be nested - a $LIB file can contain $LIB directives.

Format
```
$LIB "filename.ext"
```

Example
See examples for $INCLUDE.

See also
$INCLUDE
$USE

16.19 $LIST, $NOLIST, $EJECT

Description
These directives control listing file output.

$NOLIST disables output to the Listing File.

$LIST resumes sending output to the listing file or printer following a $NOLIST directive.

No statements after a $NOLIST directive are not sent to the listing file, until a $LIST directive is encountered.

$EJECT forces a new page in the listing file. It is ignored if $NOLIST is in operation.

Tips:
- The creation of List Files can be enabled or disabled from Project Managers Options dialog box, "Build / Create Listing Files (.lst)".
  This will make the build slightly faster.
- You can turn off $NOLIST so that all lines are listed by using Project Manager's Option "Build / Disable $NOLIST".
  This is useful during development so that error lines inside unlisted sections are shown in the Listing File.

Format
```
$LIST
$NOLIST
$EJECT
```

See also
$TITLE, $STITLE

16.20 $NOXINIT .. $ENDNOXINIT

Description
Delimits uninitialized Texts and Data Blocks in Extension Memory.
This directive is useful only for older PCD types which can contain EPROM or Flash user program memory, such as the PCD1.M1xxx.
It is not needed by new PCD types (PCD3 etc) which restore the Extension Memory segment in a different way.

Texts and DBs 4000..7999 are stored in Extension Memory, which is always RAM (read/write).
Code and Texts/DBs 0..3999 can be stored in EPROM or Flash memory (read-only).
If the battery backup for Extension Memory fails, then Texts/DBs 4000..7999 are lost.

To prevent this problem, an "Extension Memory Initialization Segment" (EMI) can be programmed into EPROM or Flash in an unused area of Text memory. The EMI segment is used to restore Extension Memory data. The EMI segment is created from the Text and DB definitions in the user program.

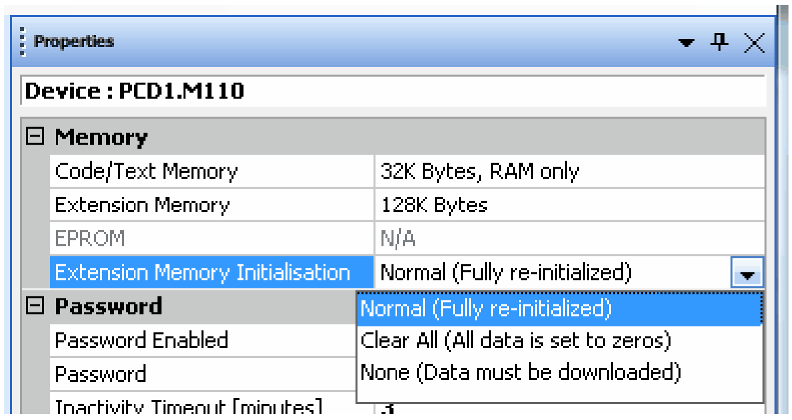
The $NOXINIT directive stops the initialization data being stored in the EMI segment, only Text/DB numbers and sizes are stored, even if the Text/DB is defined with data. If a Text/DB in Extension Memory is restored without initialization data, a Text will contain all spaces and a DB will contain zeros.

If $NOXINIT is used, then initialization data values must be downloaded from the PCD file into Extension Memory using the Download Program dialog box "Extension Memory" option.

These directives have no effect on Texts/DBs 0..3999 which are stored in Text memory.

Note
If the PCD type supports the "Extension Memory Initialization Segment", then you will see an option in the Device Configurator called "Extension Memory Initialization".
This has three settings: Normal, Clear All and None.
"Normal" means that the creation of the initialization segment is controlled by the $NOXINIT directive.
"Clear All" creates an EMI segment with NO initializer data for any Text/DB, this is the equivalent of $NOXINIT for every Text/DB in Extension Memory, all DB values are set to zero and Texts are set to

spaces. "Clear All" is useful if PCD memory is full and there's not enough space for the EMI segment.
"None" means that no initializer data is stored, all Texts are set to spaces and all DB values are set to zero.


Properties
Device : PCD1.M110
Memory
Password

Code/Text Memory	32K Bytes, RAM only
Extension Memory	128K Bytes
EPROM	N/A
Extension Memory Initialisation	Normal (Fully re-initialized)
Password Enabled	
Password	
Inactivity Timeout [minutes]	3

Normal (Fully re-initialized)
Clear All (All data is set to zeros)
None (Data must be downloaded)

Format
```
$NOXINIT
    ...        ;Text and DB definitions (plus any other code)
$ENDNOXINIT
```

16.21 $ONERROR

Description
Defines a message which is displayed in Project Manager's "Messages" window immediately after the error message when an assembler error occurs. The text also appears in the listing file. The message text remains valid until the next $ONERROR statement. If there is no text after $ONERROR then the message is removed.

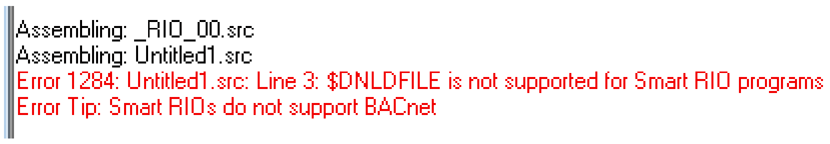
The message can contain expressions and Macro parameter references, see Using symbols in $directives.

The message text is shown in red if it begins with the word "Error", otherwise it is shown in the default text colour, usually black.

Note
$ONERROR is processed by the assembler. It does not work for errors detected by the linker. For example, if the error is caused by an invalid External symbol, then $ONERROR will not be processed.

Format
```
$ONERROR [message]
```

Example
```
$ONERROR Error Tip: Smart RIOs do not support BACnet
$DNLDFILE MyFile.5bn,0x10
$ONERROR              ;No OnError text
...
```


```
Assembling: _RIO_00.src
Assembling: Untitled1.src
Error 1284: Untitled1.src: Line 3: $DNLDFILE is not supported for Smart RIO programs
Error Tip: Smart RIOs do not support BACnet
```

See also
$ERROR
$FATAL
$WARNING
$REPORT
Using symbols in $directives

16.22 $PCDVER

Description
This directive defines the required version of PCD firmware. Use this if the program requires a particular feature which is available only in certain firmware versions.

The directive has two formats. The first is for the firmware versions of the older non "NT" systems, where each PCD model's had a different range of firmware versions and the version number was two or three digits, e.g. 003. The second format is for the more recent "NT" systems, where all PCD models use the same firmware version numbering, and the firmware version is three numbers separated dots, e.g. 1.16.32

When the program is downloaded, the destination PCD's firmware version is checked and a warning is given if it is not compatible with the $PCDVER directives in the program.

New Format, for NT systems
```
$PCDVER min_version[,max_version]
```

`min_version`	The minimum required firmware version. A warning will be issued if the PCD contains an earlier version.
`max_version`	Options. The maximum firmware version. A warning will be issued if the PCD contains a later firmware version.

A symbol name can also be used for the version number, but it cannot be a External symbol. The symbol's value must be 9 digits, aaabbbccc. For example, for firmware version 1.16.32 the symbol should be defined with the value: `001016032`.

Example
```
$PCDVER 1.16.32    ;requires firmware version 1.16.32 or later.

MinFWVersion EQU 001016032
$PCDVER MinFWVersion
```

Old Format, for old PCD systems
```
$PCDVER type version [internal_version]
```

`type`	The PCD type, as displayed on S-Bug's title line. The last digit should always be 'x' because this cannot currently be read from the PCD. The type is compared with the PCD firmware version read from the PCD. If they match, then version and `internal_version` (optional) are compared.
`version`	The earliest official firmware version for the defined PCD type upon which this program will run. Must be 3 characters, e.g. 004. To prevent the program running on a particular PCD type, use NONE, e.g. to prevent the program being loaded into a PCD1 use: `$PCDVER PCD1.M1xx NONE`

`internal_version`	This is optional. Preliminary PCD firmware is often released with an internal version number, such as B3A, $41 etc. If this is present then the last 2 characters of the internal version number are also checked. `internal_version` must be 2 characters, it excludes the first character (usually 'B', 'X' or '$'). Examples: 3A, 41.

Example values for `type`:

Type	**PCD models**
PCD2.M1xx	PCD2.M110/M120/M150/M170 etc
PCD4.Mxxx	All PCD4s
PCD6.M54x	PCD6.M540
PCD6.M2xx	PCD6.M210/M220/M230/M250/M260
PCD6.M1xx	PCD6.M100 etc

Examples

```
;On a PCD6.M1xx, the f/w version must be V004 or above
$PCDVER PCD6.M1xx 004

;On a PCD6.M540, the f/w version must be V003 or B2A or above
$PCDVER PCD6.M54x 003 2A

;The program cannot be run on a PCD6
$PCDVER PCD6.Mxxx NONE
```

16.23 $REPORT

Description
Outputs a message to Project Manager's "Messages"window, followed by a linefeesd.
This directive is useful for displaying messages at specific points during a long assembly, or can be used inside $IF..$ENDIF statements to indicate which conditional code is being generated etc.
The message is output when the $REPORT directive is processed during pass 2 of the assembly.

The message begins from the first character following $REPORT which is not a space or a tab. If no message is present, a line feed is output to the console.

The message can contain symbols and expressions, see Using symbols in $directives.

The @STR() and @ATTR() special operators can also be used, so you can display strings and symbol attributes.

Format
```
$REPORT message
```

See also
$ONERROR

16.24 $SASI .. $ENDSASI

Description
These directives are used to delimit texts which are used by the SASI instruction (Assign Serial Interface).

These SASI Texts are fully checked by the assembler and invalid texts are detected.
The texts are also converted to upper case as required by SASI.
If $SASI..$ENDSASI is not used, it is possible to enter an invalid text which may cause incorrect initialization of the serial port.

Tip: SASI Texts which contain Register indirect references, e.g. $Rnnnn, are not checked as carefully as Texts which do not contain register indirect references.
To be sure, first enter the Text with direct references and build the program to check it, then replace the direct references with $Rnnnn.

Format

```
$SASI
SASI_text_definitions
...
$ENDSASI
```

Example

```
SASI   0         ;Initialize serial port 0
       100       ;using Text 100
...
$SASI            ;Text 100 is checked by assembler
TEXT 100 "UART:9600,7,E,1;MODE:MC0;DIAG:F1000,R4000;"
$ENDSASI
```

See also
SASI
SASI Texts

16.25 $SFPARAM .. $ENDSFPARAM

Description
These directives are used to create a format definition of System Function parameters.
They are used when creating a System Function (SF) library, each library has a file ".lib" which defines the parameters for each System Function in the library.

System Functions with $SFPARAM definitions are shown in the IL Editor's "Function Selector" window, from where the SF call can be copied into the IL code.

The SF parameters are also validated by the build according to the $SFPARAM definition.

Each parameter is defined with a name, type, direction and comment, as described below. If the parameter is an array, the array size must also be given.

Format

```
$SFPARAM library_name, function_name, number_of_params
    type ['['array_size']'] direction [;comment]
    ...
$ENDSFPARAM [library_name]
```

`library_name`	The symbol name of the SF library, this is used in the CSF instruction.
`function_name`	The symbol name of the parameter. This symbol may be used inside the FB code. It is not normally needed by the caller of the FB.
`type`	The data type of the parameter. In many cases more than one type is supported. This is indicated by an | (or) character between the types. These are the currently supported data types:

```
R
T
C
I
O
F
K
DB
TEXT
X
T|C
R|T|C
R|K
R|F
DB|X
DB|X|R
X|K
DB|K
DB|X|K
X|R
DB|R
X|R|K
DB|R|K
DB|X|R|K
F|K
ANY       Any type including untyped 16-bit
BOOL      Same as I|O|F
I|O|F     Same as BOOL
UINT      16 bit unsigned
INT       16-bit signed
FLOAT     R containing Motorola FFP value
IEEE      R containing IEEE float value
```

`array_size`	If the parameter is an array, its size must also be given. This is usually used only for Register and Flag types.
`direction`	Indicates if the parameter is read or written (or both) by the FB.

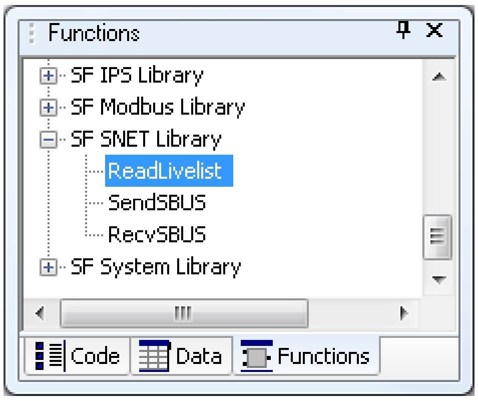
`IN`	Parameter is read, can be any data type.
`OUT`	Parameter is written, cannot be a constant data type, e.g. K, INT, UINT, FLOAT, IEEE etc.
`INOUT`	Parameter is read and written, cannot be a constant data type, e. g. K, INT, UINT, FLOAT, IEEE etc.

Examples

This is the library file for the S-Net system functions, from file SFSnetLib_en.lib.
This also defines the library version and all symbol names are in the `S.SF.SNET` group.

```
;SFSnetLib

PUBL __LIBVERS__._SAIA_SFSNET
__LIBVERS__._SAIA_SFSNET    EQU 002000001   ;SF SNet Library version

$GROUP S.SF.SNET                            ;SF SNet library

    Library                 EQU     4       ;Library number

    ReadLivelist            EQU     0
    ReadLivelist_nParams    EQU     2
    SendSBUS                EQU     1
    SendSBUS_nParams        EQU     7
    RecvSBUS                EQU     2
    RecvSBUS_nParams        EQU     7

    $SFPARAM Library, ReadLivelist, ReadLivelist_nParams
        K IN                ;Interface to be used
        R|F OUT             ;copies the Live list in a diagnostic buffer
    $ENDSFPARAM

    $SFPARAM Library, SendSBUS, SendSBUS_nParams
        K IN                ;Interface to be used
        X|R|K IN            ;IP or FDL address of the remote station
        R IN                ;S-BUS address of the remote station
        K IN                ;Number of media to send
        ANY IN              ;Base address of elements in the master PCD
        ANY IN              ;Base address of elements in the slave PCD
        R OUT               ;Returned Value
    $ENDSFPARAM

    $SFPARAM Library, RecvSBUS, RecvSBUS_nParams
        K IN                ;Interface to be used
        X|R|K IN            ;IP or FDL address of the remote station
        R IN                ;S-BUS address of the remote station
        K IN                ;Number of media to send
        ANY IN              ;Base address of elements in the master PCD
        ANY IN              ;Base address of elements in the slave PCD
        R OUT               ;Returned Value
    $ENDSFPARAM

$ENDGROUP S.SF.SNET
```

This is used by S-Edit to show the library in its "Function Selector" window:



When the function is added to the IL code, a template is inserted which can be filled in with the actual parameters:

```
    CSF     S.SF.SNET.Library       ;Library number
            S.SF.SNET.ReadLivelist  ;SF Function
                                    ;1 K IN, Interface to be used
                                    ;2 R|F OUT, copies Live list to buffer
```

See also
CSF

16.26 $SKIP .. $ENDSKIP

Description
All statements between the $SKIP and $ENDSKIP directives, or all statements after $SKIP until the end of the file, are skipped (ignored) by the assembler.
The ignored section does not appear in the Listing File.

These directives can be used to delimit very long comments or descriptions, or to temporarily patch out sections of code.

The $IF 0 .. $ENDIF conditional directives (0 is always false) can be used to delimit sections without preventing them from appearing in the Listing File.

Tip: $SKIP cannot be used inside a $IFxxx..$ENDIF conditional block unless $ENDSKIP appears before the $ENDIF, because the $ENDIF will be skipped.

Format
```
$SKIP
...          ;statements to be skipped
$ENDSKIP
```

Example
```
$SKIP
This example has been skipped.
$ENDSKIP
```

See also
$IF

16.27 $SERIALNO

Description
Defines the serial number of the PCD to which the program belongs. A warning will be shown if an attempt is made to download the program into a PCD with a different serial number.

Only one $SERIALNO directive is allowed per program. The serial number can also be a symbol name.

Format
```
$SERIALNO number
```

Examples
```
$SERIALNO 0FFA5h

SerialNumber EQU 047FA2H
...
$SERIALNO SerialNumber
```

See also
$STATION
$IPADDS

16.28 $STATION

Description
Defines the destination S-Bus station number for the program.
A warning will be shown if an attempt is made to download the program into any other S-Bus station.

This directive should be used to ensure greater security on S-Bus networks.

Format
```
$STATION stn
```

Example

```
;Allow this program to be downloaded only into S-Bus station 11
$STATION 11
```

See also
$IPADDS
$SERIALNO

16.29 $TITLE, $STITLE

Description
$TITLE provides the title which appears on the second line of every page of the Listing File, and is also used when generating Documentation files.

$STITLE provides the subtitle which appears on the third line of every page of the listing.

The title or subtitle text begins from the first character following the directive which is not a space or a tab, and ends at the end of the line.
The maximum length of a title or subtitle text is 70 characters, characters after this are ignored.
If no text appears after the directive, any existing title or subtitle is removed.

The title appears on each page of the listing, the subtitle appears on all pages following the one in which the $STITLE directive appears.
If more than one $TITLE appears, the last one in the module is used.

The text can contain symbols and expressions, see Using symbols in $directives.

Format
```
$TITLE [text]
$STITLE [text]
```

16.30 $USE, $IFUSED, $IFNUSED

These are advanced directives which is not normally needed for user programs.

Description
The normal $IF conditional directives work only with local symbols or absolute values, they do not work with external symbols defined in other files because the type/value must be known when the file is assembled (externals are only available to the linker – and there is no 'conditional linking'). $USE and $IFUSED allow the use of an external symbol (defined in another fille) in a conditional statement.

Example
In File1.src:

```
    $USE symbol, "File2.src"
```

In File2.src:

```
    $IFUSED symbol    ;(or $IFNUSED)
    ...
    $ENDIF
```

Details

$USE defines a symbol name and the name of a file. This causes the file to be assembled and linked, and the symbol is accessible to $IFUSED and $IFNUSED directives in the file.

In the above example it causes File2.src to be assembled and linked, and the `symbol` is defined and visible to $IFUSED directives in File2.src. `symbol` is a special symbol which is only visible to $IFUSED and $IFNUSED directives in file File2.src, and it cannot be used anywhere else.

When $USE is found in a file, S-Asm adds the file name to the end of the list of files to be assembled, and adds the symbol to a special symbol table which can be accessed only by $IFUSED and $IFNUSED directives in the $USE file.

$IFUSED is true if the symbol has been defined with $USE.
$IFNUSED is true if the symbol has **not** been defined with $USE.

The default extension for the file is '.src', but any extension can be used, e.g. '.srx' for an encrypted file – see notes below.
The file name can also contain a relative path or directory tag, e.g. <$LibsDir>.
If no path is given, the directories are searched in the same order as for $include files.

These directives will often be used to exclude the creation of a block if the block is never called.

For example, in file Main.src:

```
    ...
    $USE    MyBlock, "LibFile.src"
    CFB     MyBlock
    ...
```

In file LibFile.src:

```
    ...
    $IFUSED MyBlock
    FB    MyBlock
    ...
    EFB
    $ENDIF
```

Notes

- **Warning:** If you have a $USE file that includes other files, the $USE file will not be re-assembled automatically if you edit the $include file because S-Asm will not notice the change. You must do "Rebuild All".
- Files containing $IFUSED and $IFNUSED cannot be assembled and linked as program files. They can only be referenced by $USE.
- $ELSEIFUSED and $ELSEIFNUSED are also supported.
- $USE and $IFUSED/$IFNUSED cannot be used in the same file. The file which contains $USE file cannot contain $IFUSED or $IFNUSED directives.
- $USE can be inside a conditional directive or Macro.
- Encrypted source files:
  $USE will often be used with libraries (e.g. FBox libraries). Library developers may not want their code distributed in a form which can be viewed and edited by others. Source files and include files

can now be encrypted using the MAKESRX application. Encrypted files can be assembled or $included just like a normal file, and can also be used with the $USE directive. Once encrypted, the file can be assembled, but can never be viewed or un-encrypted, and no listings will be produced. Encrypted files must have one of these extensions: .srx .syx .inx .lix .dex, because S-Asm uses the extension to recognize an encrypted file.

- The same $USE directive can appear several times in a file or in different files. The $USE file is assembled and linked only once (like a $LIB file).
- The same $IFUSED directives can also appear several times in a file (like $IF).
- $IFUSED directives can be nested as for the normal $IF directives.
- The $USE file can have code outside $IFUSED sections. The code is always assembled if there is a $USE directive for the file.

16.31 $WARNING

Description
Displays a user-defined warning message in green in Project Manager's "Messages" window, and increments the warning count.
This would normally be used inside a $IF..$ENDIF statement, which can be used to detect the warning.
Warnings do not stop the build.

The warning message can contain expressions and macro parameter references, see Using symbols in $directives.

The @STR() and @ATTR() special operators can also be used, so you can display strings and symbol attributes.

Format
```
$WARNING warning_message
```

Example
```
$IFNDEF Axis
$WARNING "Axis" is not defined
$ENDIF
```

If Axis is not defined, then the assembler generates a warning:
```
Warning 6: AXIS.SRC: Line 32: "Axis" is not defined
```

See also
$ERROR
$ONERROR
$FATAL
$REPORT

16.32 $WRFILE

Description
Writes a text line to a file during the assembly process. The file is created when the first $WRFILE is assembled, or if the file already exists its length is set to zero. The file is closed at the end of assembly.

Format
```
$WRFILE "path" any_text
```

Details

The `path` is the name of the file to be created or appended to. It must be enclosed in double quotes "...". If is has no directory name, then the default directory is the device directory. If the path contains a directory which does not exist, it will be created.

The `path` can also contain these directory tags:

`<$ProjectsDir>`	Directory containing the Projects, without a trailing '\'
`<$ProjDir>`	Directory of the current Project, without a trailing '\'
`<$CpuDir>`	Directory of the current CPU, without a trailing '\'
`<$Fname>`	Name of the file being assembled, without the path or extension
`<$CpuName>`	Name of the device

For example, to create a ".txt" file with the same name as the file being assembled, in a subdirectory of the device directory called "Logs", use this as the path: "<$CpuDir>\Logs\<$Fname>.txt"

The `any_text` string is written to the file followed by LF/CR.

The string can contain expressions, symbols or macro parameters between @..@ in the same way as for $ERROR, see Using symbols in $directives. The text between @..@ characters is evaluated as a typed expression after macro parameter substitution. To allow macro parameter substitution but not evaluate it as an expression, put the character '&' immediately after the first @, e.g. `@&...@`. This causes the parameter name to be replaced by the actual parameter without evaluating it as an expression, see the example below. To display a single @, use @@.

The text generated by an expression or symbol between @..@ can be formatted as a floating point number with a given number of decimal places. This works for actual floating point constants, and for integers which have an assumed decimal point position,. For example, the integer value 1234 could be formatted as 12.34 (2 decimal places). To do this, use the `".nP"` format, when n is the number of decimal places. For example: @1234.2P@ would be output as 12.34. If a symbol `MySymbol` with the value 1234 was used it would have the same result: @MySymbol.2P@.

The @STR() and @ATTR() special operators can also be used, so you can write strings and symbol attributes to the file.

When the file is first opened, a message is displayed in Project Manager's message window:

```
Assembling: Untitled1.src
Opening $WRFILE: c:\Projects\MyProject\Log\Logfile.txt
...
```

Example

The path, or part of it, can also be a macro parameter, by enclosing the parameter name between: `@&...@`

For example, in a macro definition :

```
    WrFileMacro MACRO path, string
    $WRFILE @&path@ @&string@
    ENDM
```

Macro call:

```
    WrFileMacro("Logfile.txt", String to be written)
```

This macro call is expanded as:

```
    $WRFILE "Logfile.txt" String to be written
```

See also

Using symbols in $directives
@STR()

@ATTR()
Strings

16.33 $XOBSEG .. $ENDXOBSEG

Description
These directives are similar to the existing $INIT..$ENDINIT directives which put code into XOB 16, except that code between $COBSEG..$ENDCOBSEG is put into a COB, and code between $XOBSEG..$ENDXOBSEG is put into an XOB. $XOBSEG 16 is the same as $INIT.

If the block is already defined in the user program, the code between these directives is inserted at the start of the block, before the code that is already defined.
If the block is not already defined in the program, it is created and added to the end of the user program.

These directives can be nested up to 10 deep, i.e. a $COBSEG can contain a $COBSEG which can contain a $XOBSEG or $INIT segment and so on to a depth of 10.

For $COBSEG, the `cob_number` is optional. If $COBSEG has no `cob_number` parameter, then the code between $COBSEG and $ENDCOBSEG is placed in an automatically allocated COB which is added to the end of the user program. This COB will be executed cyclically after all preceding COBs have been processed.
`cob_number` or `xob_number` define the number of the block into which the code will be inserted. This number must be an absolute value or symbol, it cannot be an external or dynamically allocated symbol.

The code is inserted by creating a PB containing all the code, and inserting a call to this PB at the start of the COB or XOB. Every COB and XOB has a NOP inserted at the start, and S-Asm replaces this with CPB to call $xxSEG code.

Format
```
$COBSEG [cob_number]
...
$ENDCOBSEG

$XOBSEG xob_number
...
$ENDXOBSEG
```

Example
```
    MyCob   EQU COB 10
    ...
    $COBSEG MyCob
    CFB     DoSomeStuff     ;code for COB MyCob
    $ENDCOBSEG

    $XOBSEG 16              ;same as $INIT
    LD      R 120           ;code for XOB 16
            0
    $ENDXOBSEG

    COB     MyCob
    LD      R 123
            456
    ECOB
```

This code is generated:

```
    COB     MyCob
    CPB     299           ;inserted by S-Asm
    LD      R 123
            456
    ECOB

    ;This is inserted by S-Asm
    PB      299
    CFB     DoSomeStuff
    EPB
```

See also
$INIT..$ENDINIT

16.34 Using symbols in $directives

Description
The text in the $TITLE, $STITLE, $REPORT, $ERROR, $ONERROR, $WARNING and $WRFILE directives can contain Macro parameter references and expressions.

To delimit an expression from the normal text, it must be enclosed between @...@ characters, e.g. `@FRED+1@`. Macro parameters are replaced and the expression is evaluated before the text is output, and the @...@ characters are removed. This allows the types and values of symbols to be shown.

To replace macro parameters, but not evaluate the expression, place an & character (ampersand) immediately after the first @, e.g. *`@&param@`*`.` This replaces *`param`* with the macro parameter but does not evaluate the expression, so that the actual macro parameter text is output rather than its value.

To place an actual @ character in the text, use @@.

The @STR() and @ATTR() special operators can also be used, so you can display strings and symbol attributes, these do not need to be placed between @..@.

Example 1
```
FRED  EQU R 10
TOM   EQU FRED+1
...
$REPORT TOM is @TOM@, same as @FRED+1@
```

This displays the text:
```
TOM is R 11, same as R 11
```

Example 2
```
Symbol0 EQU STR "Hello, world"
...
$REPORT The string symbol is "@STR(Symbol0)"
```

This displays the text:
```
The string symbol is "Hello, world"
```

See also

Using Symbols in Texts
@STR
@ATTR
STR Strings

17 @ Operators

"Special operators" perform special calculations on their parameters. The calculations are done at assembly or link time, not when the program is executed. All special operators begin with @, followed by the operator name, and the parameters enclosed in brackets (...). There should be no spaces between the @ and the opening bracket.

A special operator can be used anywhere that a symbol or constant can be used.

`@ADDS()`	Returns the PCD firmware address
`@ARRAYSIZE()`	Returns the number of elements in an array
`@ATTR()`	References a symbol's attribute
`@ATYPE(), @NTYPE()`	Returns a value representing a variable's data type
`@CHK()`	Checksum of Text or DB
`@DFPHI() and @DFPLO()`	For loading 64-bit IEEE Double values into 2 Registers
`@IEEE()`	Convert to IEEE Float
`@IFP() and @FPI()`	Float to Integer and Integer to Float conversion
`@IFPE() and @EFPI()`	Integer to IEEE Float, and IEEE Float to Integer conversion
`@IPADDS()`	Converts an IP address to an integer
`@ISFLOAT()`	If it is a floating point value (Motorola FFP or IEEE)
`@ISIEEE()`	If it is an IEEE Float value (not Motorola FFP)
`@LEN()`	Length of Text or DB
`@MPTR()`	Get media pointer
`@POW()`	power, x to the power of y (x ^ y)
`@STR()`	References a string, @STR(..) is replaced by the actual string
`@STRLEN()`	Returns the length of a string

17.1 @ADDS( ) - Returns the media address in PCD internal format

Description
Converts the expression into the PCD firmware address format - the same format as an instruction operand.
This 16-bit address can be used by special firmware for directly accessing the data value.

The expression can be symbol or an absolute address with one of these types: I O F T C R TEXT DB

@ADDS() can be used anywhere, and the expression can contain an external reference.

NOTE
This address CANNOT be used in a user program, it is only for defining addresses which are processed by the firmware.
Does not support the new 16-bit address format.

Format
```
@ADDS(expression)
```

Examples

```
    AddsFlag1  EQU  @ADDS(F 1)  ;AddsFlag1 = 4001H (16385)
    EXTN ExtnReg
    LD   R 0
         @ADDS(ExtnReg)

    MyReg EQU  R 123
    LD   R 0
         @ADDS(MyReg)       ;81ECH (33260)

    DB 100 [3]   @ADDS(F 1), @ADDS(ExtnReg), @ADDS(MyReg)
```

The returned PCD address is a 16-bit value, and includes the data type in bits 15..13.
Register addresses are also multiplied by 4.
In the examples above it will be loaded into the lower 16 bits of the Register or DB element.

Tip: Do not use @ADDS() to get a value which is the item's number (e.g. R 123 = 123) because it does not return the correct value.
Instead, you can use a symbol name and load the value straight into a Register with LDL/LDH (the type is ignored).
Or you can use a 'K' prefix to convert the symbol into a K constant.

For example:

```
    MyReg EQU R 123     ; a symbol name must be used
    LDL   R 0
          MyReg         ; loads R 0 with 123
    ADD   R 0
          K MyReg       ; uses K 123 (the 'R' type is removed)
          R 0
```

17.2 @ARRAYSIZE( ) - Returns the size of an array

Description
If the symbol is an array, `@ARRAYSIZE` returns the number of elements in the array.
If symbol is not an array the return value is 0.
`@ARRAYSIZE` can be used in expressions in the same way as a decimal number.
The symbol must not be declared externally.
An error is generated if the symbol is external or is a label.

Tip:
This has been superseded by the new _ArraySize_ symbol.
For every symbol which is defined as an array, another symbol is automatically generated which is assigned the size of array, for example:

```
    Symbol EQU R 10
```

This generates the internal symbol:

```
   _ArraySize_.Symbol EQU 10
```

This symbol can also be made Public:

```
   PUBL _ArraySize_.Symbol
```

Format

```
@ARRAYSIZE(symbol)
```

Examples

```
Symbol1 equ F [100]
Symbol2 equ DB [10]
Symbol3 equ R 1

Len1    equ @arraysize(Symbol1)  ;Len1 = 100
```

```
Len2    equ @arraysize(Symbol2)  ;Len2 = 10
Len3    equ @arraysize(Symbol3)  ;Len3 = 0, Symbol3 is not an array

Val1    equ @ARRAYSIZE(Symbol1) + @ArraySize(Symbol2)
```

See also
ArraySize symbol

17.3 @ATTR( ) - Returns a symbol's attribute string

Description
Symbols can be assigned an *attribute string* using the $ATTR directive. The symbol's attribute string can be accessed as assembly time using the `@ATTR( )` operator, which is replaced by the attribute string before the line is assembled. `@ATTR( )` can also return a symbol's group or sub-group name, or the group nesting length. The attribute string does not have to be enclosed in double quotes "...".

There are some pre-defined attribute names, they are not case-sensitive:

`Type`	Returns the data type, e.g. "F" "R" "TEXT" "DB" etc
`Value`	The symbol's value, e.g. "1200"
`Expression`	The symbol's expression, e.g. "FBase+1"
`Comment`	The symbol's comment, without the semi-colon, e.g. "Destruct Button"
`Name`	The symbol name without the group name
`Group` *	The symbols group name, e.g. S.PRJ.Device1.IPChannel => S.PRJ.Device1, see examples below
`SubGroup,start,e`	Returns the group name from sub-group level 'start' to 'end', see examples below
`NumGroups` *	Returns the number of group names as a numeric string, e.g. "Group0.Group1.Symbol" returns "2"

* From PG5 V2.1.300 ($2.1.260)

Format
`@ATTR(`*`symbol_name, attribute_name`*`)`

Examples

The attribute string does not have to be enclosed in quotes:

```
    $ATTR ROOM=Level 7, Room 101
    MySymbol EQU R 123  ;This is the comment
    ...
    $REPORT @ATTR(MySymbol, ROOM)
```

Output is:

```
Level 7, Room 101
```

This pre-defined attribute outputs the comment:

```
    $REPORT The comment for MySymbol is: @ATTR(MySymbol, Comment)
```

If the attribute string is enclosed in quotes, they are not removed. This can be used for defining a TEXT:

```
    $ATTR MyAttribute="The attribute's string"
    MySymbol EQU R 123
    ...
    TEXT 5000 "", @ATTR(MySymbol, MyAttibute)
```

Text 5000 becomes this:

```
    TEXT 5000 "", "The attribute's string"
```

Take care with quotes!

Note that @ATTR( ) returns the string without the quotes. If you need the text to be in quotes (which you will, unless the string is a symbol name or the string already has the quotes as in the previous example), then the @ATTR( ) statement must be inside the double quotes, for example:

```
    MySymbol EQU R 123 ;This is MySymbol's comment
    ...
    TEXT 5001 "Comment is: @ATTR(MySymbol, Comment)"   ;right
    TEXT 5002 "Comment is:", @ATTR(MySymbol, Comment)  ;WRONG!
```

The result is:

```
    TEXT 5001 "Comment is: This is my symbols comment"   ;right
    TEXT 5002 "Comment is:" This is my symbols comment    ;WRONG!
                        ^
    Error: Symbol not found: This
```

Group names

The `Group` attribute can be used to return the entire group name:

`@ATTR(Group0.Group1.Group2.Symbol, Group)` is replaced by `Group0.Group1.Group2`

The `SubGroup` attribute can be used to return one or more sub-group names:

`@ATTR(Group0.Group1.Group2.Symbol, Group, 1, 2)` is replaced by `Group1.Group2`

`@ATTR(Group0.Group1.Group2.Symbol, Group, 0, 0)` is replaced by `Group0`

The start/end group numbers can be symbols or expressions, see the example below.

The `NumGroups` attribute returns the number of groups:

`@ATTR(Group0.Group1.Group2.Symbol, NumGroups)` is replaced by `3`

This can be used with the `Group` attribute by defining a symbol with the group length:

```
    Num DEF @ATTR(Group0.Group1.Group2.Symbol, NumGroups) - 1 ;Num has the value 2
```

`@ATTR(Group0.Group1.Group2.Symbol, Group, Num, Num)` is replaced by `Group2`

These examples make more sense if you imagine the symbol name is a macro parameter or an FBox parameter.

The result can used to generate a new symbol in the same group:

```
    @ATTR(param1, Group).NewSymbol EQU R
```

See also

$ATTR

@STR( )

17.4 @ATYPE( ), @NTYPE( ) - Returns the data type (ASCII or numeric)

Description

These return a value which represents the data type of a symbol or Macro parameter.
They work for R T C I O F TEXT DB DBX IB, but not for code block types like PB, FB etc.

@ATYPE() returns an ASCII value and @NTYPE() returns a numeric value.

The variable tested can be a symbol or absolute value, it cannot be an external or a dynamic address.

They are particularly useful in macros for determining the type of a macro parameter.

Notes

- 16-bit, 32-bit and 13-bit K constants all return the same values, 0 or ' '.
- Variables with type T or T|C both return the same @ATYPE value 'T', but the @NTYPE values are different (T=18, T|C=4).
- Variables with type I or I|O both return the same @ATYPE value 'I', but the @NTYPE values are different (I=20, I|O=2).

Type	**Type**	**@ATYPE result**	**@NTYPE result**
None (constant)		' ' (space)	0
K constant	`K`	' ' (space)	0
Label		'L'	1
Input/Output	`I|O`	'I'	2
Input	`I`	'I'	20
Output	`O`	'O'	21
Flag	`F`	'F'	3
Timer/Counter	`T|C`	'T'	4
Timer	`T`	'T'	18
Counter	`C`	'C'	19
Register	`R`	'R'	5
Text	`X, TEXT`	'X'	14
Semaphore	`S, SEMA`	'S'	15
Data Block	`DB`	'D'	17
Extended Data Block	`DBX`	'B'	23
Information Block	`IB`	'N'	24

Format

```
@ATYPE(expression)
@NTYPE(expression)
```

Examples

```
Sym1  EQU  R 10
Sym2  EQU  @ATYPE(Sym1)    ;Sym2 is the ASCII 'R'
Sym3  EQU  @NTYPE(Sym1)    ;Sym3 is the integer
Sym4  EQU  @ATYPE(R 123)   ;Sym4 is the ASCII 'R'

$if @ATYPE(Sym1) = 'R'     ;if symbol is a register
...
$endif
```

See also

$IFTYPE

17.5 @CHK( ) - Checksum of Text or DB

Description
Returns the checksum of a Text or Data Block. The checksum is the modulo-256 sum of all the bytes in the Text or DB.
The checksum is the same as that used by some of the PCD communications protocols, and can be used for calculating the checksums of pre-defined messages.
It can be used in the same way as @LEN().

Notes
- The @LEN() and @CHK() special operators can be used in a Text, DB or DBX, but only if the symbol they reference in not an external or dynamic address, and the Text or DB they reference does not contain any external data.
- The special operators @LEN and @CHK are converted to internal symbols by the assembler, and these symbols are resolved by the linker after the Text or DB which they reference has been processed.
  These symbols look like this: __LEN__X100, __CHK_P01_MyText. The first 6 characters are from the special operator name, __LEN__ = @LEN, the rest are from the symbol or absolute value and offset.

Format
```
@CHK(expression)
```

17.6 @DFPHI( ) and @DPFLO( ) - Separate IEEE Double into DWORDs

Description
Returns the upper or lower 32 bits of a 64-bit IEEE Double value. IEEE Double values take two Registers. These operators can be used to load the Registers with a constant or symbol value.

Symbol names cannot be assigned to IEEE Double values because a symbol's value is only 32 bits. But @DFPHI() and @DFPLO() will convert a 32-bit symbol - either an IEEE float or Motorola Fast Floating Point (FFP) value - into a n IEEE Double and return the upper or lower 32 bits. This allows you to define 32-bit symbols and use them as 64-bit IEEE doubles - but note that the range and precision is limited if you use symbols. If a constant is used the range and precision is not limited.

Format
```
@DFPHI(value) ;returns upper DWORD of IEEE double value, e.g. @DFPHI(1.2345678)
@DFPLO(value) ;returns lower DWORD of IEEE double value, e.g. @DFPLO(1.2345678)
```

Examples
@DFPHI( ) and @DFPLO( ) also accept IEEE or FFP symbols, their values are converted to double:

```
IEEESymbol EQU 1.2345678I    ;with 'I' postfix for IEEE float
FFPSymbol  EQU 1.2345678

;converts IEEESymbol to IEEE Double and returns the lower 32 bits
IDoubleLO  EQU @DFPLO(IEEESymbol)

;converts IEEESymbol to IEEE Double and returns the upper 32 bits
IDoubleHI  EQU @DFPHI(IEEESymbol)

;converts FFPSymbol to IEEE Double and returns the lower 32 bits
FDoubleLO  EQU @DFPLO(FFPSymbol)

;converts FFPSymbol to IEEE Double and returns the upper 32 bits
FDoubleHI  EQU @DFPHI(FFPSymbol)
```

To declare an IEEE double directly you can use an IL Macro like this:

```
;Load 2 registers with an IEEE double value
;'value' can be a symbol (IEEE float or FFP), or a constant
DFLD MACRO reg, value
     LD R reg
        @DFPHI(value)
     LD R reg+1
        @DFPLO(value)
ENDM
...
Symbol1 EQU 1.234I ;IEEE float value
Symbol2 EQU 1.234  ;FFP value

DFLD(R 0, Symbol1) ;Loads R 0..1 with 1.234 IEEE Double from IEEE float
DFLD(R 2, Symbol2) ;Loads R 2..3 with 1.234 IEEE Double from FFP value
DFLD(R 4, 1.234)   ;Loads R 2..3 with 1.234 IEEE Double from the constant
```

See also
Floating Point Instructions

17.7 @IEEE( ) - Convert to IEEE Float

Description
Converts a string, decimal, hex or Motorola FFP value to IEEE float. The conversion is done at build time. The value can be a symbol, number, expression, string, macro parameter etc.

Format
```
@IEEE(value)
```

Examples
```
;Result         Expression
123             DecSymbol    EQU  123
03F9D70A4H      HexSymbol    EQU  3F9D70A4H    ;1.23 IEEE float in hex
1.23I           IEEESymbol   EQU  1.23I
1.23            FFPSymbol    EQU  1.23
                String       EQU  STR "1.23"

123.0I          Sym0         EQU  @IEEE(DecSymbol)    ;convert decimal to IEEE
123.0I          Sym6         EQU  @IEEE(123)          ;convert decimal to IEEE
1.23I           Sym1         EQU  @IEEE(HexSymbol)    ;hex value as IEEE
1.23I           Sym2         EQU  @IEEE(03F9D70A4H)   ;hex value as IEEE
1.23I           Sym2         EQU  @IEEE(IEEESymbol)   ;already IEEE, no conversion
1.23I           Sym3         EQU  @IEEE(FFPSymbol)    ;convert FFP to IEEE
1.23I           Sym4         EQU  @IEEE(@STR(String)) ;convert string to IEEE
1.23I           Sym5         EQU  @IEEE(1.23)         ;convert immediate to IEEE
```

See also
@IFP( ) - Integer to IEEE Float
@ISIEEE( ) - Is it an IEEE Float value
Floating Point Instructions

17.8 @IFP( ) and @FPI( ) - Integer to FFP Float and FFP Float to Integer

Description
Converts a Motorola floating point value to an integer or vice-versa. Can be used to convert Macro parameters.

Format
```
@IFP(int_value, exponent)     ;returns FFP float value: int * 10^exponent
@FPI(ffp_value, exponent)     ;returns an int value from FFP float
```

Note: There are also these versions for IEEE floating point numbers:
```
@IFPE(int_value, exponent)    ;returns IEEE float value: int * 10^exponent
@EFPI(ieee_value, exponent)   ;returns an int value from IEEE float
```

Example
```
;Load Register 100 with FFP floating point value 1.23
LD      R 100
        @IFP(123, -2)   ;123 * 10^-2 = 1.23
```

See also
IFP, FPI instructions
@IFPE( ) and @EFPI( ) Integer to IEEE float and IEEE float to integer
Floating Point Instructions
@ISFLOAT( )
@ISIEEE( )

17.9 @IFPE( ) and @EFPI( ) - Integer to IEEE Float and IEEE Float to Integer

Description
Converts an IEEE floating point number to an integer or vice-versa. Can be used to convert Macro parameters.

Format
```
@IFPE(int_value, exponent)    ;returns IEEE float value: int * 10^exponent
@EFPI(ieee_value, exponent)   ;returns an int value from IEEE float
```

Example
```
;Load Register 100 with IEEE floating point value 1.23
LD      R 100
        @IFPE(123, -2)
```

See also
@IFP() and @FPI() for Motorola fast Floating Point (FFP)
Floating Point Instructions
@IEEE( ) - Convert to IEEE Float

17.10 @IPADDS( ) - Convert IP address to integer

Description
Converts an IP address with the format `a.b.c.d` to a 32-bit integer value. Where `a`, `b`, `c` and `d` are constants 0..255 or symbols with the value 0..255.

Notes
- If a symbol is used, it cannot be External.
- Symbols cannot contain group names because it conflicts with the '.' in the IP address.

Format
```
@IPADDS(a.b.c.d)
```

Examples
```
Symbol1 EQU @IPADDS(1.2.3.4)        ;result in hex is 001020304H

Adds3   EQU 2
Symbol2 EQU @IPADDS(128.15.1.Adds3) ;result in hex is 0800F0102H

;IP addresses can also be added (interesting but probably not very useful ;-)
Symbol3 EQU @IPADDS(1.2.3.0) + @IPADDS(0.0.0.4) ;result is 001020304H
```

17.11 @ISFLOAT( ) - Is it an FFP or IEEE Float value?

Description
Returns 1 if the item is a floating point value (Motorola FFP or IEEE), or 0 if it's not. This useful for testing macro parameters.

Format
```
@ISFLOAT(value)   ;value can be a symbol or numeric value
```

Example
```
$IF @ISFLOAT(Symbol)
...
$ENDIF
$IF @ISFLOAT(1.2)
...
$ENDIF
```

See also
Floating Point Instructions
@ISIEEE( )

17.12 @ISIEEE( ) - Is it an IEEE Float value?

Description
Returns 1 if the value is an IEEE floating point value, or 0 if it's not. This is useful for testing macro parameters.
IEEE floating point values end with an '`I`', e.g. `1.2I`, `10I`, this distinguishes them from FFP floating point values, which have a different binary format.

Format
```
@ISIEEE(value)   ;value can be a symbol or numeric value
```

Example
```
$IF @ISIEEE(Symbol)
...
$ENDIF
$IF @ISIEEE(1.2I)
...
$ENDIF
```

See also
Floating Point Instructions

@ISFLOAT( )
@IEEE( ) - Convert to IEEE Float

17.13 @LEN( ) - Length of Text or DB

Description
Returns the length of a Text or Data Block. Text lengths are in characters (excluding the terminating NUL, if present). Data Block lengths are the number elements in the DB.

Notes
- The @LEN() and @CHK() special operators can be used in a Text, DB or DBX, but only if the symbol they reference in not an external or dynamic address, and the Text or DB they reference does not contain any external data.
- The special operators @LEN and @CHK are converted to internal symbols by the assembler, and these symbols are resolved by the linker after the Text or DB which they reference has been processed. These symbols look like this: __LEN__X100, __CHK_P01_MyText. The first 6 characters are from the special operator name, __LEN__ = @LEN, the rest are from the symbol or absolute value and offset.

Format
```
@LEN(expression)
```

Example

```
    MyText EQU TEXT 100
    TEXT MyText   "12345"
    TEXT MyText+1 "123"
    LenMyText EQU @LEN(MyText)    ;=5
    LenMyText1 EQU @LEN(MyText+1) ;=3

    LD     R 0
           @LEN(MyText)        ;R 0 = 5
    LD     R 0
           LenMyText           ;R 0 = 5
    LD     R 0
           @LEN(MyText) + 4    ;R 0 = 5 + 4 = 9
    LD     R 0
           @LEN(MyText+1)      ;R 0 = 4
    ...
    DB 0 [2]  @LEN(DB 0), 3   ;Error, cannot be used
                               ;in Texts or DBs
```

See also
@CHK() Checksum of Text or DB
@STRLEN() Returns the length of a String

17.14 @MPTR( ) - Get Media Pointer

Description
Returns the 32-bit media pointer address of the media item (absolute address or symbol). This allows symbols to be created with media pointer addresses. External or dynamic symbol names cannot be used.

Format
```
@MPTR(absolute_media_expression)
```

Examples
```
;Get the media pointer to Register 1 using absolute addresses
MediaPtrR1 EQU @MPTR(R 1)

Symbol1 EQU R 1
Mptr1 EQU @MPTR(Symbol1)

;These two examples do the same thing
XLA  R 0
     DB 4000
LD   R 0
     @MPTR(DB 4000)

;Dynamic or external addresses are not allowed
Symbol2 EQU R
Mptr2 EQU @MPTR(Symbol2)  ;ERROR! dynamic address not known by assembler
EXTN Symbol3
Mptr3 EQU @MPTR(Symbol3)  ;ERROR! external value not known by assembler
```

See also
Media pointer instructions
XLA Load address

17.15 @POW( ) - Power (x ^ y)

From PG5 V2.1.300 ($2.1.261)

Description
Returns the result of x to the power of y, where x and y are both integers, both FLOAT or both IEEE.
Data types cannot be mixed.
No error occurs on overflow or underflow.

Values of x and y	Return value of @POW( )
x < > 0 and y = 0.0	1
x = 0.0 and y = 0.0	1
x = 0.0 and y < 0	INF

Format
```
@POW(x, y)
```

Examples
```
IntX   EQU 12
IntY   EQU 2
IntZ   EQU @POW(IntX, IntY)      ;IntZ = 12^2 = 144

FloatX EQU 12.0
FloatY EQU 2.0
FloatZ EQU @POW(FloatX, FloatY)  ;FloatZ = 12.0^2.0 = 144.0

IeeeX  EQU 12.0I
IeeeY  EQU 2.0I
IeeeZ  EQU @POW(IeeeX, IeeeY)    ;IeeeZ = 12.0I^2.0I = 144.0I

FloatX EQU 12.0
```

```
IeeeY  EQU 2.0I
IntZ   EQU @POW(FloatX, IeeeYY)  ;Error! incompatible data types
```

See also
@IFP() and @FPI() - Convert integer to Motorola FFP fast floating point and back
@IFPE( ) and @EFPI( ) - Integer to IEEE floating point and back
IEEE float

17.16 @STR( ) - References a string

Description
A *string* is not a Text (as in Texts and Data Blocks), it is a sequence of characters which can be inserted into the IL code in a similar way to a macro parameter. But unlike macro parameters, they can be used anywhere in the file, inside or outside a macro. Some new FBox Adjust parameters are *strings* - not symbols or values, but simply some textual information.

Strings can only by referenced using @STR( ). @STR( ) can be used directly in the directives for text output, $REPORT, $WARNING, $WRFILE etc. It is not necessary to enclose this operator in @...@ characters to enable it to be evaluated, using `@STR( )` alone is the same as using: `@&$STR( )@`. @STR( ) operators are resolved after macros have been expanded, and before the code is assembled. This allows string names to be passed as macro parameters.

Defining a string
Strings defined with a string name and the data type STR, followed by the string's text in double quotes "...". When the string is referenced using the @STR( ) operator, the quotes are removed. String symbol names are kept in a separate symbol table, so their names will not clash with normal symbol names. String symbol names are valid from to point of definition to the end of the file, forward references are not allowed.

```
String100 DEF STR "some text"
```

If you want to keep the quotes, use double double quotes as in this example, @STR( ) removes only the outer quotes.

```
MyString DEF STR ""Keep the quotes""   ;@STR() removes only the outer quotes
```

Notes
- If an error occurs when processing @STR( ), then any other @STR( ) or @ATTR( ) operators on the same line will not be processed, and will generate a "syntax error".
- If the @STR( ) string is empty, and you have enclosed it in @...@ because it is in a $directive, for example:
  `$REPORT @@STR("")@`
  then it resolves to @@, which outputs a single @ character. In this case the solution is to remove the outer @...@ :
  `$REPORT @STR("")`
- String names are not affected by $GROUP directives.
- For macro parameters, you can either pass the @STR(...) operator as the parameter, or pass the string or string name and reference it with @STR() inside the macro.

Format
```
@STR("string")
@STR(string_name)
```

Examples
```
MyString EQU STR "Strings"
```

```
...
$WRFILE "Test.txt" @STR(MyString) are fantastic
```

Result written to file Test.txt is:

```
Strings are fantastic
```

See also

Strings

@ATTR( )

17.17 @STRLEN( ) - Gets the length of a String

From PG5 V2.1.300 ($2.1.260)

Description

Returns the length of a string excluding the quotes. Returns -1 if the *`string_name`* is empty.

Tip: This is useful for detecting blank macro parameters when `$IFB <string>` fails if the string may contain the '>' character.

Format

```
@STRLEN(string_name)
```

Examples

```
MyString       EQU STR "1234"
MyStringLength EQU @STRLEN(MyString)

$IF @STRLEN(MyString) = 4
  $REPORT Length is 4
$ENDIF

StringMacro MACRO param1
  $IF @STRLEN(param1) = -1
    $REPORT param1 is blank
  $ENDIF
ENDM
```

See also

Strings

@STR( )

@ATTR( )

18 Macros

Macros are the most powerful feature of the Saia PG6 IL Language. No other PLC manufacturer has an interpreted IL language which supports macros.

A macro is a block of code which is defined once, with a special name, and can be "called" many times in the program using the macro name in the same way as an instruction mnemonic.

Macros can be given parameters which are replaced by actual values when the macro code is generated. The macro can be called with different parameters which are referenced by the code inside the macro. This can cause different code to be generated.

In effect, macros can be used to define the equivalent of new IL instructions.

Whenever a macro name is found in the program, the block of text from the macro definition is inserted. Wherever a macro parameter is referenced, the parameter is replaced by the parameter supplied with the macro call. The macro call is "expanded" into the full text of the macro.

A macro is not the same as a block (FB, PB etc), because the code of a block exists only once, but the code of a macro is repeated every time it is used. But unlike the code in a block, the code generated by a macro code is usually different every time.

Macros are used extensively by Fupla, the code for every FBox is a Macro.

The Macro Examples section contains macros for Bit, Byte and Word access to Registers and Data Blocks.

Advantages and Disadvantages of Macros

Macros are faster than FBs or PBs because no parameters have to be accessed, and there is no actual "call" instruction.

Instead of repeating the same code more than once, you can define it in a macro, and call (expand) the macro several times with different parameters. For example, with a different base address.

The other big advantage with macros is that you do not need to use the Index Register or Register Indirect instructions to access data from a base address. If the base address is passed as a macro parameter, you can access it directly. E.g.

```
MyMacro MACRO ModuleBase
STH     ModuleBase+0
ANH     ModuleBase+1
ANH     ModuleBase+2
ANH     ModuleBase+3
OUT     ModuleBase+4
ENDM
```

If this was in an FB, you would need to load the Index Register with the base address and use STHX, ANHX, OUTX etc.

The only disadvantage is that more code can be generated because to code is inserted whenever the macro is called.

See also
Defining a Macro
Calling a Macro

$IFB, $IFNB - If blank, if not blank
$IFE, $IFNE - If equal, if not equal
LEQU, LDEF - Local symbols
GEQU, GDEF - Nested macro symbols
Macro Examples

18.1 Defining a Macro

Description
Macros are defined with a macro name, `MACRO` keyword, and optional parameter list, a macro body containing the macro's code, followed by `ENDM`.

Macros can have up to 250 parameters whose names can be supplied on one or more lines, separated by commas. If the macro has parameters, the name of the first parameter must be on the same line as the `MACRO` statement. Other parameters can span several lines providing there is a comma separator ',' between each parameter, and after the last parameter on each line. The parameter names are used like symbols inside the macro body. When the Macro is called, all references to the "formal" parameters are replaced by the "actual" parameters, see Calling a Macro.

The Macro body can contain any statements or directives. The `ENDM` statement ends the macro definition.
Note: `ENDM` must not be preceded by a label on the same line, e.g. `Label: ENDM` is illegal. Instead, put the label on the preceding line.

Tip: Macros are often defined in $include files, so they can be used in many source files.

Format
```
macro_name MACRO [param1] [, param2]...
    statements  [;[;]comment]]
    ...
    ENDM
```

EXITM
To end macro expansion before `ENDM` is reached, you can use `EXITM`. This could be useful inside a `$IF` statement to simplify the macro definition, see examples in $IFB, $IFNB.

Creating new symbols from macro parameters
The `#` character can be used as a delimiter between a formal parameter and other text in the macro so that symbol names or new expressions can be created, see Example 2 below.

Referencing macro parameters in Strings (STR)
From PG5 V2.1.300, macro parameters can be referenced from inside a String by using `"@&`*`param`*`@"`. Without the enclosing `@&...@`, the macro parameter '*`param`*' is not replaced. For example:
```
  DemoMacro MACRO param0, param1
  String1 EQU STR "Macro parameters are: '@&param0@' and '@&param1@'"
  $RPEORT @STR(String1)
  ENDM
```

Example 1
Macro definition
```
;AND gate macro
ANDGATE MACRO INPUT1, INPUT2, OUTPUT  ;macro name and parameters
    STH     INPUT1        ;;local comment
    ANH     INPUT2        ;comment
    $IFNB < OUTPUT >      ;if macro parameter OUTPUT is not blank
    OUT     OUTPUT
    $ENDIF
ENDM                      ;end of macro
```

Macro call

```
ANDGATE(I 0, I 1, F 2)
```

Expands to this code

```
    STH     I 0
    ANH     I 1           ;comment
    OUT     F 2
```

Example 2

The # character can be used as a delimiter between a macro parameter and other text in the macro so that symbol names or new expressions can be created, as in this example.

Macro definition

```
MYOBJ MACRO objname
objname#_property1 PEQU R
objname#_property2 PEQU R

LD      objname#_property1
        1
LD      objname#_property2
        2
ENDM
```

The macro call

```
MYOBJ(Object1)
```

Expands to

```
Object1_property1 PEQU R
Object1_property2 PEQU R
LD      Object1_property1
        1
LD      Object1_property2
        2
```

You can use & to get the value of the macro parameter instead of the parameter itself:

```
CreateChannel MACRO channel
Channel_#&channel EQU channel
ENDM
...
```

The macro call

```
ChannelNum EQU 10
CreateChannel(ChannelNum)
```

Expands to

```
Channel_10 EQU 10
```

See also

Calling a Macro
$IFB, $IFNB - If blank, if not blank
$IFE, $IFNE - If equal, if not equal

LEQU, LDEF - Local macro symbols
GEQU, GDEF - Global macro symbols
Macro Examples

Notes

- Macros cannot be called before they have been defined in a source file.
- Macro names can contain the same characters as symbols.
- Macro names and macro parameters cannot have the same name as any other symbol, reserved word or instruction mnemonic.
- Macro parameters can have the same name as symbols or labels defined outside the macro - the macro parameter names are all local to the macro.
- Macro definitions can contain macro calls, which can themselves contain macro calls, and so on up to a nesting depth of 9.
- Nested macro *definitions* are not allowed - macros cannot be defined within macros.
- Jump labels inside a macro are always local to the macro. Unique label prefixes are generated by the assembler. It is illegal to jump into a macro from outside, or to jump out of a macro. Keep local label names as short as possible.
- The DEF declaration should be used to define the names of symbols used in macros.
  If EQUate is used in a macro, a "multi-defined symbol" error occurs if the macro is called more than once in the same file.
- $INCLUDE in a macro includes the file in the macro definition, NOT in each macro call. The file is included only once, when the macro definition is processed.
- $IFDEF and $IFNDEF do not work with macro parameters. Use $IFB and $IFNB instead.
- To compare actual macro parameter strings use $IFE and $IFNE.
- Other $IFxxx..$ELSE..ENDIF statements inside a macro are treated normally. These statements can contain macro parameters.
- The expansion of macros in the listing file (.LST) can be enabled or disabled from Project Manager's "Options" dialog box.
- Macros are listed in the cross-reference list at the end of the listing file in the same way as symbols - where they are defined and where they are called is shown.
- Macro parameters cannot be special operators.
- `ENDM` must not be preceded by a label on the same line, e.g. `Label: ENDM` is illegal. Instead, put the label on the preceding line.
- Comments inside macros which are preceded by two semi-colons `;;` are not shown in the listing. They also do not take up space in memory during the build - this is from the days when PCs only had 128KB RAM, so it's obsolete now.

18.2 Calling a Macro

Description

To call a macro, the macro name is used as if it is an instruction mnemonic.
Actual parameters are supplied as a list enclosed in brackets, e.g. `(param1, param2)`, with each parameter separated by a comma.
Parameters can be on one or more lines.

When the macro is expanded, the parameter references inside the macro body are replaced by the actual parameters.
This is done by simple string replacement. The parameter name is replaced by the string supplied as the actual parameter.
The generated code is assembled after the parameters have been replaced.

If a macro has been defined to accept parameters, it is not always necessary to supply all the parameters.
The $IFB and $IFNB directives can be used to check for the existence of a parameter, and the $IFE and $IFNE directives can be used to compare a parameter string with a given string.

If parameters are left out, the correct number of commas must still be present so that the parameters are in the correct positions.
For example, in this macro call which takes 4 parameters, parameters 1 and 3 are not supplied:

```
    FRED(,param2,,param4)
```

If the last parameter (or last few parameters) will be left out, the trailing commas should still be present:

```
    FRED(param1,param2,,)
```

Notes
- Macros cannot be called before they have been defined, the Macro definition must appear first in the source file.
- Parameters are not replaced inside comments or Texts.
- Actual macro parameters cannot contain these characters: `( ) ; ,`
- Leading and trailing spaces are stripped from each actual parameter.

Format

```
macro_name ( [param1] [,param2]... ] )   [;[;]comment]
```

The brackets `( )` must be present, even if there are no parameters.
The parameters are separated by commas. Commas must always be present even if the parameter is empty.

Example
Macro call:

```
    ANDGATE (I 0, I 1, O 32)
```

Using example definition in Defining a Macro, this macro call is expanded to:

```
    STH    I 0
    ANH    I 1       ;Comment
    OUT    O 32
```

See also
Defining a Macro
$IFB, $IFNB - If blank, if not blank
$IFE, $IFNE - If equal, if not equal
LEQU, LDEF - Local macro data
GEQU, GDEF - Global macro data
Macro Examples

18.3 $IFB, $IFNB - If blank / if not blank

Description
These directives can be used to control the expansion of a macro according to the actual parameters supplied with the macro call.

$IFB and $IFNB mean "if blank" and "if not blank". These are used to determine if an actual parameter was supplied or not.

Notes
- Use $IFE and $IFNE to compare macro parameters.
- To compare symbol values use $IF or $IFN.
- To check symbol types use $IFTYPE.
- For detecting empty strings, if the string may contain '>', you can use `@STRLEN(param) = -1`.

@STRLEN() is useful for detecting blank macro parameters when `$IFB <string>` fails if the string may contain the '>' character.

Format

```
$IFB <parameter>    ;The < and > must be present!!
...
$ELSE               ;and nested $IFB's are allowed
...
$ENDIF
```

Example 1

AND gate macro.

Macro definition:

```
ANDGATE MACRO INPUT1, INPUT2, OUTPUT
    STH     INPUT1
    ANH     INPUT2
    $IFNB <OUTPUT>       ;;if parameter "OUTPUT" is supplied (not blank)
    OUT     OUTPUT       ;;then write to OUTPUT, else
    $ENDIF               ;;leave result only in ACCU
ENDM
```

Macro call:

```
    ANDGATE(I 32, I 63)
```

Expands to:

```
    STH   I 32
    ANH   I 63
```

Example 2

`EXITM` ends the expansion of the macro, this can be used inside conditional directives. This is useful to exit from deeply nested `$IFxxxx` statements. For example:

```
ANDGATE MACRO INPUT, OUTPUT
        STH   INPUT
        ANH   INPUT+1
        $IFB <OUTPUT>      ;If "OUTPUT" is blank,
        EXITM              ;stop macro expansion
        $ENDIF
        OUT OUTPUT         ;else write the output
ENDM
```

Macro call:	Expands to:

`ANDGATE (I 0)`	`STH   I 0`
	`ANH   I 1`

`ANDGATE (I 0, O 32)`	`STH   I 0`
	`ANH   I 1`
	`OUT   O 32`

See also

$IF
$IFE, $IFNE - If equal, if not equal
@STRLEN() Gets the length of a String

18.4 LEQU, LDEF - Define local macro data

Description
These declare symbols which are local to the block (COB, FB etc) or to the Macro in which the statement appears.
They are the same as the EQU and DEF statements, except they are used inside macros or blocks.

This allows symbols to be defined within macros and blocks which do not produce "multi-defined symbol" errors if the macro is called more than once in the same file, or if temporary data uses the same symbol name in different blocks within the same file.

Symbols declared with LEQU or LDEF cannot be accessed directly by any nested macros (for this you should use GEQU or GDEF), and they cannot be accessed from outside the block.

Symbols declared with LEQU and LDEF cannot be made Public.

LEQU and LDEF symbols are not affected by $GROUP, the group name is not used.

Format
```
local_symbol_name LEQU [type] [expression]  [;comment]
local_symbol_name LDEF [type] [expression]  [;comment]
```

Example

```
BigMac      MACRO  param1
;Symbols local to a macro
Sym0        LEQU   R
Sym1        LEQU   R

            INC    param1

            ENDM


            COB    0
                   0
;Symbols local to the block
Reg0        LEQU   R
Reg1        LEQU   R

            bigmac(Reg0)

            ECOB

            PB     0
Reg0        LEQU   R
Reg1        LEQU   R

            bigmac(Reg0)

            EPB
```

See also
EQU
DEF
PEQU

GEQU and GDEF

Technical Info
To create a local symbol, the assembler adds a group name to the symbol to make it unique. A different group name is used for each block and each macro expansion. The prefix begins with an underscore, so you won't normally see these symbols in the "All Symbols" or "Data List" views in SPM unless you select "Internal Symbols".
Inside a macro, the group name is __mac__xxxxxx, inside a block the group name is __lequ__xxxxxx, where "xxxxxx" is a string which is unique to each macro call and each block.

You can see the group names in the Listing files. This is the code which is generated by the above example, taken from the listing file:

```
            COB    0
                   0
            NOP                   ;inserted by S-Asm for call to init code
;Symbols local to the block
__lequ__fghb89.Reg0        LEQU   R
__lequ__fghb89.Reg1        LEQU   R

            bigmac(__lequ__fghb89.Reg0)
;Symbols local to a macro
__mac__1h8phc0.Sym0        LEQU   R
__mac__1h8phc0.Sym1        LEQU   R
            INC    __lequ__fghb89.Reg0

            ECOB

            PB     0
__lequ__x4v9mx.Reg0        LEQU   R
__lequ__x4v9mx.Reg1        LEQU   R

            bigmac(__lequ__x4v9mx.Reg0)
;Symbols local to a macro
__mac__yrr75s.Sym0        LEQU   R
__mac__yrr75s.Sym1        LEQU   R
            INC    __lequ__x4v9mx.Reg0

            EPB
```

18.5 GEQU, GDEF - Define global macro data

Description
These are the same as the EQU and DEF statements, except they are for use inside Macros, and define symbols which are local to the macro but can also be accessed by all other macros which are called from inside *this* macro (nested macro calls). This is often used inside FBoxes, which often use nested macros.

For local macro symbols use LEQU or LDEF.

GEQU and GDEF symbols are not affected by $GROUP, the group names are not used.

Format
```
global_symbol_name GEQU [type] [expression]  [;comment]
global_symbol_name GDEF [type] [expression]  [;comment]
```

Example

```
Mac1 MACRO
     LD   Reg1     ;accesses symbol Reg1 defined in macro Mac2
          100
     ENDM

Mac2 MACRO
Reg1 GEQU R 100    ;Reg1 declared
     Mac1( )       ;Reg1 can be accessed from this macro
     ENDM

     COB    0
            0
     Mac2( )
     ECOB
```

The above example generates this code:

```
     COB  0
          0
     NOP                          ;inserted by S-Asm for call to init code
     Mac2( )
__mac__g_1h8phc0.Reg1 GEQU R 100  ;Reg1 declared
     Mac1( )                      ;Reg1 can be accessed from this macro
     LD   __mac__g_1h8phc0.Reg1   ;accesses symbol Reg1 in macro Mac2
          100
     ECOB
```

19 File Formats

The build utility processes a 'make file' (.mak).
The assembler produces a 'listing file' (.lst) for each source file that is assembled.
The linker produces a single 'map file' (.map) for the program.

Make File (.mak)	Created by the build utility to control the build procedure
Listing File (.lst)	Contains details of the assembly process and code generated by macros
Map File (.map)	Contains details of the link process

19.1 Make File (.mak)

The "make file" is a text file which is passed to the build utility to control the build operation.
It contains file names and switches for controlling the assembler and linker.
Any text which is not a switch (or a comment) is assumed to be a source file name.

`;comment`	Comments are allowed in the make file. All text from then ';' to the end of the line is ignored.
`/PCD=pcdfile`	Defines the name of the .PCD file to be created. This must always be present. pcdfile can be a path name, otherwise the PCD file is created in the current directory.
`/NOLINK`	Link is not done, files are assembled only.
`/Q`	Stop the make process on the first error.

`/Dsymbol [=value]`	Defines a symbol, with an optional value in any units, for example:
	`/DFRED=1`
	`/dtito=1.2`
	`/dchar='A' /dHexVal=0ABH`
	If no value is given, 0 is used. Up to 10 symbols can be defined in this way. This symbol is often used to control conditional assembly using $IF.

`/Ipathname`	Specifies an additional include file search path. pathname is the path name of a directory which contains include files. The current or supplied directory is searched first, followed by the `/I` directories. Up to ten /I statements can be given.
`/$Ifilename`	Includes the file `filename` at the start of every source file by generating a $INCLUDE statement. Up to 10 `/$I` include files can be defined.
`/NOMAP`	Don't generate a .MAP file.
`srcfile[.src]`	Any other texts in the file (not preceded by '/' or ';') are assumed to be the names of source files to be assembled. The default .SRC extension is appended if no extension is present. `srcfile` can contain a path, but if no path is given then the current directory is assumed.
`/L`	Create listing files named "srcfile.LST". The default is no listings.
`/NM`	Don't expand macros in the listing file. Only the macro call is listed, not the code which is created by the macro. This makes listing files much smaller.
`/NG`	Don't list Graftec incoming and outgoing ST/TR parameters in the listing file.
`/NOTIT`	Produces listing files without the title lines or any pagination. This can be useful to produce a listing file which can be compared with an older listing file, because the date/time on the title lines are always different.
`/NONOLIST`	Disables all $NOLIST directives so that listing files contain ALL the source code. This may produce very large listing files.
`/FRM`	Allow forward references to macros.

	This is an advanced switch, and should not normally be used. The PG5 does not allow forward references to macros (calling a macro before it has been defined). PG4 V1.4 did allow this, and so did the PG3, so this switch allows compatibility. Using forward references to macros will cause "pass 2 phase errors" if a label is used after the macro call.
`/VOLF=adds`	Defines the last Volatile Flag address. This should be the same as the address used in the DEFVM instruction. If flags are defined with the F VOL type, S-Asm checks that the address is below or equal to this address. This switch is needed because the DEFVM instruction may be in another file and without the switch the assembler would not know the DEFVM address. `adds` should be in the range 0..8191. Omit the switch if Volatile Flags are not used.
`/WOSTV`	Warn on symbols with the same type and value, see /WOA.
`/WOSNA`	Warn on offset to symbol which is not an array.
`/WOA`	Warn on *all* symbols with the same type and value. This switch is ignored unless the /WOSTV switch is used to warn on symbols with the same type and value. Normally this check is not done on symbols beginning with an underscore. Using /WOA enables the check for all symbols.

Notes

- Many of the above switches relate directly to Project Manager's "Build Options".
  For more details, examine the '*cpu_name*.mak' file produced by Project Manager.
- The build is always done in the current directory, so you do not need to use full path names for files.
  If the files are in other directories then try to use relative paths, e.g. "..\Dir\File.inc".
- File names containing spaces must be enclosed in "double quotes", and must not wrap onto another line.
- Program Smake52.exe is a command-line version of the build utility, which processes a make file.
- Smake52 can be used in a batch file for automated builds, or invoked by other applications.

19.2 Listing File (.lst)

Listing files are only produced if "Create Listing files" is "Yes" on Project Manager's "Build Options" dialog box.
The listing file is produced even if there are assembly-time errors.
Listings are useful for examining the code generated by macros, and for examining the location of errors.
Some aspects of the listing format can be controlled using option on the "Build Options" dialog box.

The listing width is 122 characters, and must be printed either on 132 column paper, or on 80 column paper with the printer set to "compressed pitch" mode.

For unsatisfied conditionals ($IFxxx which is not TRUE), the ADDR, OPC M OPERAND and IP fields are left blank, statements within the unsatisfied block are not processed.
Statements between $SKIP..$ENDSKIP and $NOLIST..$LIST directives do not appear in the listing unless the "No $NOLIST" option is checked.

Listing file example

```
SAIA PCD MACRO ASSEMBLER $1.0.119                                                       PAGE 1
FOR SAIA'S INTERNAL USE ONLY
MODULE: Demo.src (09/20/00 13:32:48)  ASSEMBLED: 09/20/00 13:32
title

SOURCE LINE      ADDR   MNEMO  MC  OPERAND         IEM  SOURCE

0-0                                                     $include "_New.inc"
1-1                                                1    ;Generated by SAIA Project Manager
1-2                                100             1    __TIME_BASE__  EQU 100          ;timer timebase in ms
1-3                                31              1    __LAST_TIMER__ EQU 31           ;last timer number
1-4                                                1
0-0                                                     $include "_Global.inc"
2-1                                                1    EXTN    Symbol0 C
0-0                                                     $include "c:\x\708\New\Demo.sy5"
3-1                                                1    ;;This file was generated by Symbol Editor
3-2                                                1    ;;Don't edit it manually
3-3                                                1    ;{DEFSYMNAME}: Symbol
3-4                                                1    ;{DEFGRNAME}: Group
3-5                                                1    ;{DEFOPTIONS}: 4
0-1                                                     ; LISTING FILE FORMAT
0-2                                                     $TITLE  title
0-3                                                     $STITLE subtitle
0-4
0-5                                                                 EXTN   Valve
0-6                                                                 PUBL   Motor
0-7                            I   0                    On          EQU    I 0                 ; On switch
0-8                            O   33                   Motor       EQU    O 33                ; Turns on the motor
0-9                            F   100                  Error       EQU    F 100               ; Error flag
0-10                           F   101                  MotorOn     EQU    F 101               ; High is motor is on
0-11
0-12                                                    ;------ MOTOR CONTROL BLOCK
0-13             0      PB         0                                PB     0
0-14             1      STH    I   0                                STH    On                  ; If ON switch pressed
0-15             2      ANL    F   100                              ANL    Error               ; and no error flag
0-16             3      OUT    O   33                               OUT    Motor               ; turn on the motor
0-17             4      OUT    F   101                              OUT    MotorOn             ; and set the flag
0-18             5      OUT    ?   0                E               OUT    Valve               ; and open the valve
0-19             6      OUT        0                                OUT    Lamp                ; and turn on the lamp
                                                                           ^

**** Error 42: Demo.src: Line 19: Symbol not defined: Lamp
0-20             7      EPB                                         EPB
0-21                                                    ;------ END OF MOTOR CONTROL BLOCK
```

Title line

The top line of each page shows the Assembler version number, the source file name and its creation date and time, the date and time of assembly, and the page number.
The date and time are formatted according to the country, month first if US etc.

Registered user name, title and subtitle

The registered user name, and an optional title and subtitle appear on the next three lines of each listing page.
The title is generated by the $TITLE directive, the subtitle is generated by $STITLE.

LINE field

Source file number and source file line number (file-line).
The first number is the number of the source file from the FILE NUMBER AND NAMES LIST shown at the start of the cross-reference list, see below.
The second number is the line number within the file.

ADDR field

Relocatable program line number. Starts from 0 in each module listing.
To determine the actual address in the PCD's memory you must add the module's start address found in the MAP file.

MNEMO MC OPERAND field

Shows the instruction mnemonic, type/special/conditional/ channel/priority code and operand.
Externally declared operands are shown as 0, or their partial value if addition or subtraction has been performed using an external symbol.

IEM field

`I`	A number in this column indicates that the line is from an include file. The number shows the include file nesting level (1..9). If the number is 1, the line is from the first include file; if it is 2, the line is from an include file which was included by the first include file etc.
`E`	Indicates the line contains an external symbol. The actual value is not yet resolved. The operand field contains the partial value of the operand.
`A`	Automatically (dynamically) assigned symbol. The actual value is not yet resolved.
`M`	A number in this column shows that the line is from a macro. The value is the macro nesting depth (1..9).

SOURCE field
Contains users source file line, exactly as it is in the source file.
If the line is longer than 122, the line wraps around onto the next line in the listing.

Error Messages
Assembly-time error messages are formatted as shown in the listing example.
The error message shows the error number, the name of the file containing the error (which may be an include file), the source file line number of the error, an error message text and a caret (^) pointing to the position of the error on the source line.

In the case of blocks which contain no closing statement, for example FB 0 with no closing EFB or $IFxxx with no closing $ENDIF, the error message appears at the end of the file.

File Numbers and Names List
This is a list of all files which are used by this source the file.
File no. 0 is the main source file, files 1..254 are the include files.
The full path names of each file are shown. For example:

```
FILE NUMBERS AND NAMES

NO.  FILE NAME
0    c:\x\708\New\Demo.src
1    c:\x\708\New\_New.inc
2    c:\x\708\New\_Global.inc
3    c:\x\708\New\Demo.sy5
```

Cross Reference List and Symbol Table
The cross reference list and symbol table are merged into one list, called the "cross reference list".
The cross reference list always begins on a new page.

Cross-reference list example

```
    _SAIA PCD MACRO ASSEMBLER V1.0.120
    FOR SAIA'S INTERNAL USE ONLY
    MODULE: Demo.src (09/20/00 13:32:48)  ASSEMBLED: 09/20/00 13:32

    FILE NUMBERS AND NAMES

    NO.  FILE NAME
    0    c:\x\708\New\Demo.src
    1    c:\x\708\New\_New.inc
    2    c:\x\708\New\_Global.inc
    3    c:\x\708\New\Demo.sy5


    CROSS REFERENCE LISTING AND SYMBOL TABLE
```

```
SYMBOL                   TYPE    VALUE           SCOPE     CROSS REFERENCE LIST f

__abs__PB_0              PB      0               ABS       0-13#
__LAST_TIMER__                   31                        1-3#
__TIME_BASE__                    100                       1-2#
Error                    F       100                       0-9# 0-15
Lamp                             0                         0-19
Motor                    O       33              PUBL      0-6 0-8 0-16
MotorOn                  F       101                       0-10# 0-17
On                       I       0                         0-7# 0-14
Symbol0                  C       0               EXTN      2-1
Valve                            0               EXTN      0-5 0-18

Assembly complete, 0 warnings, 1 errors
```

Top line and registered_user_name
As described above, but the title and subtitle do not appear.

FILE NUMBERS AND NAMES
This is a list of all source and include files used in the project. The file number is used in the LINE field of the listing file, and also in the cross-reference listing.

SYMBOL field
The symbol's name. The list is in alphabetical order, where "_" comes before "A".

SCOPE field
EXTN means external symbol, PUBL means public symbol, DEF means defined symbol, AUTO means the address will be automatically assigned (dynamic), blank means local symbol.

TYPE field
Description of the type attribute of the symbol. Types are given where the symbol is defined (e.g. Symbol EQU I 0).

Blank	**Untyped constant**
I|O	Input or Output (share same address space)
F	Flag
T	Timer
C	Counter
T|C	Timer or Counter (share same address space)
R	Register
K	K constant
COB	Cyclic Organization Block
XOB	Exception Organization Block
PB	Program Block
FB	Function Block
SB	Sequential Block
ST	Step or Initial Step
TR	Transition
SEMA	Semaphore (for LOCK and UNLOCK)
TEXT	Text (shares same address spaces as DB)
DB	Data Block (shares same address space as TEXT)
DBX	Extended Data Block

`IB`	Information Block
`=`	Function Block parameter number
`LABEL`	Label
`MACRO`	Macro
`PREDEF`	Pre-defined (internal) symbol
`?`	The type is unknown (external or error)

VALUE field
The actual value of the symbol.
Externally declared operands are shown as 0 or their partial value if an expression has been used. '?' is shown if the value cannot be computed due to an assembly-time error.
For labels, the offset from the start of the labels code block is shown.

CROSS REFERENCE LIST field
Contains each source program file number and line number where each symbol is referenced or defined, in numerical order.
The line number where the symbol is defined is postfixed with #. Note that more than one # may appear if DEF is used.

19.3 Map File (.map)

The map file name is always the name of the absolute object file with type ".map".

The map file contains the following data:

- Revision number of linker.
- Name of absolute object file created.
- Date and time of linkage.
- The Registered User's name.
- The source file names of each module linked (not the object file names).
- Assembly date and time of each module linked.
- Code start line number for each module linked.
  This can be added to the address appearing in the listing file to give the actual address of a program line in the PCD memory.
- Code size in program lines for each file.
- Text size in bytes for each file.
- Extension memory size in bytes for each file.
- The start line of the module's $INIT segment.
- Total code size in program lines.
- Total text size in bytes.
- Total extension memory size in bytes, and the extension memory initialization segment size.
- The total number of global symbols.
- The total size of the $INIT segment in lines.
- The list of downloadable files
- A list of all global symbols, their values, defining files and a list of referencing files for each symbol.

Map file example

```
SAIA PCD LINKER $1.0.119                                              PAGE 1
FOR SAIA'S INTERNAL USE ONLY
FILE: New.pcd   LINKED: 09/20/00 13:43

        CODE    CODE    TXT/DB  EXTEN
MODULE  START   SEG     SEG     SEG     ASSEMBLY          MODULE
NUMBER  LINE    LINES   BYTES   BYTES   DATE AND TIME     NAME
--------------------------------------------------------------------------------
1       8       0       20      0       09/20/00 11:27    _New.src
2       8       5       196     0       09/20/00 13:42    il.src
3       13      0       0       0       09/20/00 11:27    module2.src
4       13      8       0       0       09/20/00 13:41    Demo.src
5       21      0       0       0       09/20/00 11:27    _Global.sy5
--------------------------------------------------------------------------------

CODE SEGMENT SIZE:      21 LINES (84 BYTES)
TEXT/DB SEGMENT SIZE:   216 BYTES
EXTENSION MEM SIZE:     0 BYTES
GLOBAL SYMBOLS:         30

DOWNLOADABLE FILES
00  ..\RIO_01\192.168.10.101.srio        Type,Number: 33,0
01  ..\RIO_02\192.168.10.102.srio        Type,Number: 33,0
02  ..\RIO_20\192.168.10.120.srio        Type,Number: 33,0
03  ..\RIO_100\192.168.10.200.srio        Type,Number: 33,0
04  ..\RIO_101\192.168.10.201.srio        Type,Number: 33,0

SAIA PCD LINKER $1.0.119                                              PAGE 2
FOR SAIA'S INTERNAL USE ONLY
FILE: New.pcd   LINKED: 09/20/00 13:43

SYMBOL     TYPE VALUE         CROSS-REFERENCE LIST (MODULE NUMBERS)

__DYNAMICS__.__FIRST_COUNTER__
                1400          1#
__DYNAMICS__.__FIRST_DB__
                1500          1#
__DYNAMICS__.__FIRST_FLAG__
                7500          1#
__DYNAMICS__.__FIRST_RAM_DB__
                3000          1#
__DYNAMICS__.__FIRST_RAM_TEXT__
                2000          1#
__DYNAMICS__.__FIRST_REGISTER__
                3000          1#
Linkage complete. 0 errors, 0 warnings.
```

20 Error and Warning Messages

20.1 Assembler Errors 1000+

These are the errors detected by the assembler during the build.

Tip: To find more details about an error, you can open the Listing file and press F4 to see the actual error line, with a caret ^ pointing to the character on the line where the error was detected.

Format
Error *number*: *file*: **Line** *line*: *description*

Where:

number	The error number. Each error message has a unique number which makes it easier to find the error in the documentation or help file.
file	The name of the source or include file where the error was detected.
line	The line number in the source or include file where the error has occurred.
description	See the error messages below.

Assembler Error Messages
For simplicity only the error numbers and texts are shown in the following list of errors, and errors with self explanatory messages are not accompanied by a detailed description.

Error 1000: No file name
No source file names in the Make file.

Error 1001: Too many parameters
Error 1002: Invalid switch
A command line switch or parameter in the make file is invalid.

Error 1003: Invalid file name
File names must be valid path names. Network paths are not supported, you must assign a drive letter for these.

Error 1004: Can't open file
Error 1005: Read error on file
Error 1006: Write error on file
For reads: the source file does not exist. For writes: the disk or file is write protected, the disk is full, the file is open in another application, or you do not have the correct access rights.

Error 1008: Out of memory
This will normally never occur unless the PC has no more virtual memory available or memory has been corrupted. Try booting the PC.

Error 1009: Line too long (max. length is 1024 chars)
IL source files and include files cannot contain lines longer than 1024 characters.

Error 1010: Too many lines

A single IL source file or include file cannot contain more than 65535 lines. Split the file into two files.

Error 1011: Invalid $INCLUDE file name: *filename*
Include file names must be valid path names. Network paths are not supported, you must assign a drive letter for these.

Error 1012: Can't open $INCLUDE file: *filename*
The $include file cannot be found. If it is a library file, check that the library has been selected from Project Manager's "Library Manager".

Error 1013: $INCLUDE file nesting too deep: *filename*
A $include file can include another $include file which includes another $include file... up to maximum of 10 nested $includes.

Error 1014: Symbol in $IF not resolved/not defined
This error is usually caused by a forward reference in a $IF statement. For $IF to work, it must know the value of any symbols it references. For example, this generates the error:

```
    $IF symbol = 1    ;error 1014
    ...
    $ENDIF
    ...
    Symbol EQU 2
```

Error 1015: Error count exceeds 100
More than 100 errors causes the assembler to abort.

Error 1016: Key file USER.KEY not found or invalid
The build cannot be done unless the PG5 package has been properly registered. Contact your supplier.

Error 1017: Recursive $INCLUDE file: *filename*
An include file cannot include itself. This may happen indirectly if one include file includes another.

Error 1018: Stack overflow
This will normally never occur unless the PC has no more stack memory available or memory has been corrupted.

Error 1020: $IF nesting too deep
$IFxxx statements can be nested up to 30 deep.

Error 1021: $ELSE without $IF
An unexpected $ELSE was found. $ELSE must be preceded by $IF.

Error 1022: Missing $ENDIF
A $IF statement has no closing $ENDIF.

Error 1023: $ENDIF without $IF
Error 1024: $ENDSKIP without $SKIP
Error 1024: Missing $ENDSKIP
Error 1025: $ENDLAN without $LAN
Error 1026: $ENDSASI without $SASI
These closing directives must be preceded by the opening directive.

Error 1027: $ELSEIF after $ELSE
$ELSEIF cannot follow $ELSE because $ELSE has no condition, (ELSE is always the inverse of the state of the preceding IF).

Error 1028: EXITM outside macro
The EXITM statement is for use inside a macro definition.

Error 1029: Multi-defined macro parameter
The same formal macro parameter name has been used more than once in a macro definition.

Error 1030: Unknown directive
The $directive is not valid.

Error 1031: Syntax error
An invalid, unknown or unexpected character or statement. Invalid use of operators, missing opening or closing parentheses etc.
Note: Special operators like `@ATTR()` and `@STR()` do not allow spaces before the `(`, e.g. `@ATTR (...)` will cause a syntax error.

Error 1032: Invalid expression/Overflow/Divide by zero
The expression contains an invalid constant (out of range or result is too big/too small), divide by zero, or is an unknown mnemonic or statement.

Error 1033: Extra character(s) on line
After processing all the valid tokens on the source line, extra characters are still present, these are not processed.

Error 1034: Missing operand
The preceding instruction requires more operands.

Error 1035: Invalid operand
The operand is unrecognizable, a more detailed error message cannot be provided.

Error 1036: Unexpected operand
The operand is not required by the preceding instruction.

Error 1037: Multi-defined label: *label*
The same jump label has been defined more than once in the same block, or the label has the same name as a symbol.

Error 1038: Label outside block
Since jump labels are local to the block in which they are defined, a label cannot be defined outside a block. Labels can address the first line of a block, providing they are on the same line as the first mnemonic, e.g. `LABEL: PB  0`.

Error 1039: Illegal SB call
Sequential Blocks can only be called from a COB, FB or PB.

Error 1040: Invalid symbol
The symbol contains invalid characters.

Error 1041: Multi-defined symbol: *symbol*
The symbol has been defined more than once. Labels cannot have the same name as a symbol. To allow both EXTN and EQU declarations for the same symbol in the same file, the EXTN declaration must be first.

Error 1042: Symbol not defined: *symbol*
The symbol or label has not been defined or declared, see Declarations.

Error 1043: Symbol not evaluated: *symbol*
Another error has prevented the symbol's value from being properly evaluated. This is usually caused by chained forward references, e.g.:

```
    Sym1 equ Sym2   ;first forward reference to Sym2
    Sym2 equ Sym3   ;second forward reference to Sym3
    Sym3 equ 123    ;Sym1 and Sym2 generate error 43
```

Error 1044: Symbol has incompatible type
Typed symbols in an expression do not have the same type, or the expression's type prefix does not match the type of the symbol(s) in the expression. Also occurs if the symbol is defined as a macro.

Error 1045: Illegal use of typed symbol
A symbol defined with a type cannot be used in this context. An untyped constant should be used.

Error 1046: Invalid type
The medium type or the symbol's type is invalid.

Error 1047: Already declared external
Error 1048: Already declared public
The symbol has more than one EXTN, PUBL or PEQU declaration.

Error 1049: Labels can't be public
Labels are always local to a file. They cannot be accessed from other files (or other blocks).

Error 1050: DEFined symbols can't be public
The value of DEFined symbols can change within the source file. The assembler would not know which value to make public.

Error 1051: Illegal use of external
- Externals cannot appear in conditional directives, and cannot appear in expressions which do more than add or subtract a constant from an external, or add an external to a constant.
- $ and $$, and symbols defined with $ or $$, cannot be used in $INITsegments because the $INIT segment location is unknown and so is the $ or $$ address.
- The pre-defined symbol _BLOCKNUM_ cannot be used if the block number is external.
- Jump labels cannot be external.
- DB and Text sizes, in square brackets, cannot be external.
- FB parameter numbers can't be external.
- Symbols referenced by DEF or LDEF cannot be external, e.g. Sym DEF ExternalSym.
- DBX numbers cannot be external.
- Externals cannot be used for $COBSEG and $XOBSEG numbers.
Tip: Make the symbol local to remove this problem (use Symbol Editor's 'Make Local' command). Or, if it must be global, then select Symbol Editor's 'Use Local Declaration' option - this puts and EQU statement into the globals include file instead of an EXTN statement.

Error 1052: More than one external reference
An expression can contain only one external symbol, or symbol with a dynamic address.

Error 1053: Missing symbol
A symbol is missing from a declaration.

Error 1054: FB param numbers (=) can't be public
If Function Block parameter numbers are defined with symbols (using `symbol EQU = n`), these symbols cannot be made PUBLic. Instead, define the FB name and its parameter numbers in an include file, and include it in each file which calls the FB.

Error 1055: DBX or IB symbols can't be public
Symbols defined with DBX (Extended Data Block) or IB (Information Block) types cannot be made public. The linker cannot handle these data types.

Error 1056: Too many FB parameters (max. is 255)
An FB can have up to 255 parameters (1..255).

Error 1057: Symbol is not an array
Dynamically allocated addresses must be assigned as arrays if they are to be used with offsets, for example:
```
    Sym1    EQU F
    Sym2    EQU Sym1+1  ;Error 1057, gets the same value as sym3
    Sym3    EQU F       ;gets address sym1+1
```
To make this work, sym1 must be an array:
```
    Sym1    EQU F [10]
```
Arrays cannot be defined from arrays, for example, this does not work:
```
    BaseArray  EQU R 100[10]
    Array1     EQU BaseArray[5]   ;Array1 = R 105, it is not an array
    Array2     EQU Array1+5[5]    ;Error 1057: Symbol is not an array
```

Error 1058: Too many include files (max. 254)
Up to 254 different include files can be used in a single project. This includes nested include files.

Error 1059: Illegal use of type or condition code
A data type, MOV instruction data type or a condition code cannot be used in this context.

Error 1060: Invalid label
Invalid characters in a label name.

Error 1061: Illegal use of label
A label is illegal in this context.

Error 1062: Label not defined
The label is not defined or is not in the current block.

Error 1063: Multi-defined block
The COB, XOB, PB, FB, SB, IST, ST or TR has already been defined in this source file.

Error 1064: Block within block
Definitions of COB, XOB, PB, FB and SB code blocks cannot appear within a code block.

Error 1066: Missing end of block statement
Each code block must have an end of block statement, (COB..ECOB, XOB..EXOB, PB..EPB, FB..EFB, SB..ESB, ST..EST, TR..ETR).

Error 1067: Not in Sequential Block
Step (ST) and Transition (TR) definitions can appear only inside a Sequential Block (SB).

Error 1068: Wrong end of block statement
The end of block statement does not match the start of block statement, e.g. COB..EXOB.

Error 1069: Instruction(s) outside block
All instructions must be within a code block (COB, PB etc). Inside an SB, all instructions must be inside step or transition blocks (IST..EST, ST..EST or TR..ETR).

Error 1070: Invalid FB parameter reference

The FB parameter reference (using '=') can be used only with instructions inside an FB, or the instruction does not allow parameter references.
The LD instruction does not all an FB parameter as the value - itrequires a 32-bit operand, and FB parameters are only 16 bits. Instead, use LDH and/or LDL or pass the 32-bit value in Register.

Error 1071: Missing ENDM
A macro definition must be terminated by ENDM. ENDM cannot be preceded by a label.

Error 1072: Missing macro parameter
A formal parameter in the macro definition is missing.

Error 1073: Too many macro parameters
Macros can have up to 255 parameters.

Error 1074: ENDM without MACRO
ENDM can only be used to end a macro definition.

Error 1075: Macro call nesting too deep
A macro can call a macro, which in turn can call another macro up to a nesting depth of 9.

Error 1076: Nested macro definition
Macros cannot be defined inside macro definitions. Each macro must be defined separately.

Error 1077: Recursive macro call
A macro calls itself. This may be an indirect call via another macro.

Error 1078: Illegal use of macro
The macro name cannot be used in this context. For example, macro names cannot be made public.

Error 1079: Missing < or >
For the $IFB and $IFNB directives, the string must be enclosed in angle brackets <...>.

Error 1080: Unexpected text
A text definition inside double quotes "..." is not related to a TEXT statement.

Error 1081: Missing text
A TEXT statement is not followed by any text, or the text ends in a comma ',' so the assembler was expecting a formatted symbol.

Error 1082: Multi-defined text
The TEXT has already been defined.

Error 1083: Missing closing quote (")
A TEXT definition has no closing double quote before the end of the line.

Error 1084: Invalid character in "< >"
The character is not <, >, " or a decimal number. Numeric characters inside square brackets can be between <1> and <255>, <0> (NUL) is only allowed in Texts 4000 or above.

Error 1085: Invalid character in text
The ASCII NUL character (0) is not allowed in Texts 0..3999 because this character is used to delimit the end of the Text. NUL can only be used in Texts 4000 and above.
Note: In the GSM character set (see $CHARSET GSM), the '@' character has the value 0, and can only be used in Texts 4000 and above.

Error 1086: Text too big

The text is longer than the length given in square brackets, or the text is longer than 3072 (3K) bytes (including the terminating NUL).

Error 1087: Missing closing bracket '>'
Error 1088: Unexpected closing bracket '>'
Each opening < must have a closing > in the text. < and > characters must themselves be enclosed in angle brackets to enter these into a text: <<> <>>.

Error 1089: Invalid text number
Text numbers can be between 0 and 3999, or 4000 to 7999 if in Extension Memory. The PCD3 supports Texts/DBs up to 8191.
Note: For old PCD6 firmware versions V002 or below, text numbers are limited to 0-999.

Error 1090: Invalid text length
The text length given in square brackets is invalid, max. 3072 characters.

Error 1091: Invalid LAN text
Text within $LAN..$ENDLAN is an invalid LAN text. (obsolete)

Error 1092: Invalid SASI text
The text between the $SASI..$ENDSASI directives is not a valid SASI instruction text.

Error 1093: Missing $ENDSASI or $ENDLAN
A $LAN directive was found while inside a $SASI section, or vice-versa.

Error 1094: Invalid format
The format of a symbol in a text is invalid.

Error 1095: Invalid LAN text number (0..3999 only)
The LAN2 co-processor can only access texts 0..3999, it cannot access texts 4000..7999 in extension memory. (obsolete)

Error 1096: DBX initialization data too big
A text or data value in a DBX is too big to fit in the number of bytes assigned with the @size value, e. g.

```
@1: 0FFFFH   ;max value allowed here is 1 byte, 0FFH
```

Error 1097: Multiple use of <text> or <db> insertion
Only one insertion per Text or DB is allowed.

Error 1098: Unknown character set
Invalid character set with $CHARSET, use ANSI, OEM or GSM.

Error 1099: Offset on dynamically allocated block is illegal: symbol
If a code block is defined with an automatically allocated address, then another symbol cannot be defined with an offset from this symbol, for example:

```
  DynPB   equ pb
  DynPB1 equ DynPB+1  ;error: offset on dynamically allocated block!
```

Error 1100: Illegal use of reserved word
Indicates an attempt to use a reserved word (mnemonic, declaration or media type, condition code or MOV instruction data type, etc) as a symbol.

Error 1102: Invalid condition code
The condition code is invalid.

Error 1103: Invalid data size
The MOV instruction’s data size is invalid.

Error 1104: Incompatible data types
Symbols with different data types cannot be used in the same expression. For the STXM and SRXM instructions, the source and destination medium types must be compatible. For example, a register source cannot have a flag destination. In a MOV instruction, the data sizes of the 2nd and 4th operands must be the same.

Error 1105: Missing type
The instruction requires a medium type, or the symbol used in the operand is an untyped symbol. The operand range has not been checked. This error can also occur if a Graftec Step or Transition created by the PG4's Graftec editor does not contain any code.

Error 1107: Missing accu status
The instruction requires an accumulator status (H L P Z N E C).

Error 1108: Missing analogue channel number
The instruction requires an analogue channel number (0..3).

Error 1109: Invalid counter channel
Invalid counter channel number, see Help.

Error 1110: Invalid memory flag
Accepts 0 or 1.

Error 1111: Invalid I|O number
Error 1112: Invalid output number
The maximum number of I/Os is dependent on the PCD model. refer to your hardware documentation.

Error 1113: Invalid flag number
Old PCD models support Flags 0..8191. From firmware version 1.14.00 this was increased to 0..14335.
PCDs with firmware version 1.20.00 support 0..16383 flags, but this must be enabled from the Build Options "Use 16-bit addressing".

Error 1114: Invalid timer/counter number
The Timer/Counter partition can be manually selected from Project Manager's "Build Options", see the 'Last Timer' setting. Timer/Counter addresses are 0..1599.

Error 1115: Invalid register number
Old PCD models support registers 0..4095. The newer "NT" systems (PCD3 etc) support 0..16383. For firmware versions before 1.20.00, the Register Indirect instructions (CPBI, SASII, SCONI, SRXMI, STXMI, TFRI) cannot use Register addresses above 8191. If the error is from a symbol with an absolute address, change it to use an address <= 8191. If it's a dynamic address, open the Build Options and change the dynamic address range for Registers to start below 8191 and do a "Rebuild All". If the register's address is still assigned above 8191, try changing the link order of the files which fail, putting these at the beginning, and do a "Clean Files" operation to re-assign the dynamic addresses.
Firmware versions 1.20.00 and above allow the full range of Register addresses 0..16383 in all Register Indirect instructions - but this must be enabled from the Build Options "Use 16-bit addressing" setting. This creates a PCD file containing code in a new format, whcih is not compatible with older PCD firmware.

Error 1116: Invalid constant
Error 1117: Negative constant

A "K" constant is a 14-bit value stored in the PCD's 16-bit operand. The upper 2 bits of the operand define the data type. A "K" constant is used only in operands which require a data type (R/F/T/C/I/O/K etc). These errors can occur for K constants which or not in the range 0..16383 (0..3FFFH).
The LDL and LDH instructions load 16-bit values, 0..65535.
Other constants and symbol values, and values for the LD instruction, are 32 bit values, range -2147483640 . .2147483647.

Error 1118: Invalid PB number
Error 1119: Invalid FB number
Error 1120: Invalid COB/XOB number
Error 1121: Invalid SB number
Error 1122: Invalid ST/TR number
Error 1123: Invalid nibble number
Error 1124: Invalid bit number
Error 1125: Invalid byte number
Error 1126: Invalid word number
Error 1127: Invalid long word number
Error 1128: Invalid digit number
Error 1129: Invalid timebase
Error 1130: Invalid base address
Error 1131: Invalid semaphore
Error 1132: Invalid test number
Error 1133: Invalid exponent
Error 1134: Invalid FB parameter
Error 1135: Invalid relative address
Error 1136: Invalid accu status
Error 1137: Invalid number of elements
Error 1138: Invalid channel
Error 1139: Invalid station number
Error 1140: Invalid switch
Error 1141: Invalid control signal
Error 1142: Invalid analogue channel
Error 1143: Invalid priority
Error 1144: Invalid data block number
Error 1145: Invalid data block length
The operand is out of range or the number is invalid. See Data Types for valid ranges. Some ranges are dependent on the PCD type or FW version.
Some instructions do not allow the full range of element addresses or values, and the range of some operands is dependent on the values of preceding operands (SRXM, STXM, TFR, BITI, BITO, DIGI, DIGO etc).

Error 1146: Multi-defined data block
The Data Block has already been defined.

Error 1147: Missing data block length
Data Blocks must have a length enclosed in angle brackets, even if it's empty "[]".

Error 1148: Missing value(s)
A value was expected. Usually caused by a trailing comma in a list of values.

Error 1149: SASI or LAN texts can't have length
Special texts defined between the $SASI..$ENDSASI and $LAN..$ENDLAN directives cannot be given a [length].

Error 1150: FB parameter has bad context
The same Function Block parameter number has been used for two different operand types, one of

which is an error. Only the second use of the parameter generates an error.

```
STH  = 3      ;Parameter 3 is I|O|F|T|C element
...
BITI = 3      ;Parameter 3 is untyped 16-bit value
       ^
*** Error 150: FB parameter has bad context
```

This error often occurs when using constants. Some instructions require a K constant (ADD, CMP etc), and some require a constant without a K (e.g. LDL, LDH). The K is needed if the constant will be used in an instruction where the parameter could also be R, T, C etc. It is not possible to use the same constant as an FB parameter if it is used with both K and no-K instructions. For example, this will not work:

```
    CFB   0
          100   ;has no K type
    ...
    FB    0
    LDL   R 0
          = 1   ;OK
    CMP   R 0
          = 1   ;error! a K is required
```

This also fails:

```
    CFB   0
          K 100 ;has the K type
    ...
    FB    0
    LDL   R 0
          = 1   ;K type can only work if the K is removed
    CMP   R 0
          = 1   ;OK - but the "bad context" error is given here
```

Note that if a K constant is used only with LDH or LDL, then S-Asm will remove the K media type from the FB parameter, and no error will be generated.

Error 1151: Text already in use as data block
Error 1152: Data block already in use as text
Texts and Data Blocks share the same addressing, a Text cannot have the same number as a DB. For example, if TEXT 0 is defined then DB 0 cannot be defined, and vice versa.

Error 1153: Text/DB data too long
An explicit size has been given to a text or data block, and the number of initialisers is longer than the size. If this is a Fatal Error, then the initialiser buffer is full (holds up to about 12'900 DB element initialisers, so this error should be rare!).

Error 1154: Register indirect, must have data type
For the TFRI, STXMI and SRXMI instructions, which use register indirect addressing, the source and destination data type must be explicitly defined. For example:

```
    STXMI 0
          R 10
          I SourceReg    ;These are Registers
          F DestReg      ;e.g. SourceReg EQU R 10
                         ;the data type *must* be present
```

Error 1155: Register indirect, symbol must be a register
For the TFRI, STXMI and SRXMI instructions, which use register indirect addressing, the source and destination symbols must be registers. See above.

Error 1156: $ is illegal in $INIT and $xxxSEG segment
The $ value (offset from start of block) cannot be used inside these segments because the address is

not known.

Error 1157: JPx instruction illegal in $INIT and $xxxSEG segment
The JPI and JPD instructions cannot be used in these segments

Error 1158: Wrong $ENDINIT/$ENDxxxSEG directive
The closing directive does not match the opening directive, e.g. $COBSEG is closed by $ENDXOBSEG.

Error 1159: Missing XOB number
The $XOBSEG directive requires a specific XOB number.

Error 1160: Already in $INIT/$xxxSEG segment
Nested $INIT, $COBSEG or $XOBSEG directives are not allowed.

Error 1161: $ENDINIT/$ENDxxxSEG without $INIT/$COBSEG/$XOBSEG
Missing opening $INIT, $COBSEG or $XOBSEG directive.

Error 1162: Missing $ENDINIT/$ENDCOBSEG/$ENDXOBSEG
Missing closing statement.

Error 1163: Illegal in $INIT/$xxxSEG segment
Code in these segments is placed in a block and therefore cannot contain any block start or end instructions such as COB/ECOB, PB/EPB etc.

Error 1164: LEQU or LDEF outside macro or block
These declarations are only for use inside macros or blocks for defining symbols local to the macro or block.

Error 1165: ***user defined error message***
This error is generated by the $ERROR directive in the user program.

Error 1166: Local symbol has same name as macro parameter
Formal macro parameters in a macro definition cannot have the same name as a symbol local to the macro defined with LDEF or LEQU or as a local label.

Error 1167: Forward reference to macro
Macros must be declared in the source file before they are called, not afterwards. Move the MACRO definition or $INCLUDE directive to the start of the source file.

Error 1168: Volatile Flag address not in Volatile Flag segment
The F VOL address is after the Last Volatile Flag address defined in Project Manager's "Build Options".

Error 1169: Local symbols cannot be PUBLic
Symbols defined with LEQU are local to the macro or block in which they are defined, and they cannot be made public.

Error 1170: Multi-defined $AUTO type: *type*
The $AUTO directive is generated by Project Manager from the dynamic address ranges in the Build Options, so this error will occur if you try to use this directive in a user program.
The element type already has a $AUTO directive. Only one $AUTO is allowed per type.

Error 1171: Dynamic address overflow for type: *type*
The dynamically allocated address has gone outside the range defined on the Project Manager's "Build options". The dynamic address range must be increased.

Error 1172: No $AUTO directive for this type: *type*
The $AUTO directive is generated by Project Manager from the dynamic address ranges in the "Build Options", so this error will occur if you try to use dynamic allocation if it is turned off from the device's "Build Options".
A symbol has been declared with a dynamic address, but no $AUTO directive has been found to define the dynamic address range for this type.

Error 1173: Illegal use of dynamically allocated symbol
Symbols whose values are dynamically allocated cannot be used in DEF or LDEF declarations.

Error 1175: Dynamic address allocation not allowed for type: *type*
Only these types can be dynamically allocated: Media: R, T, C, F, TEXT (or X), DB, SEMA (or S).
Code blocks: COB, SB, FB, PB, ST, TR.

Error 1176: $AUTO directive overlap (*type / type***)**
The $AUTO directive is generated by Project Manager from the dynamic address ranges in the "Build Options", Texts and Data Blocks share the same address space, as do Timers and Counters. Because of this, the $AUTO directives for these types must not overlap, they must define separate addresses.

Error 1177: Array bounds overflow: *symbol*
This error is generated if a symbol is defined which references an array but is outside the array bounds. E.g.
```
Regs  EQU R [10]    ;Array, defines Regs+0..Regs+9
QReg  EQU Regs+10   ;Regs+10 is outside range 0..9
```

Error 1178: Multi-defined $CPU number
More than one $CPU directive exists in the source file and included files. This is only allowed if they all define the same CPU number.
Note: $CPU is now obsolete, new PCDs can have only one CPU.

Error 1179: Multi-defined $STATION number
More than one $STATION directive exists in the source file and included files. This is only allowed if they all define the same S-BUS station number.

Error 1180: Invalid $PCDVER data
The $PCDVER directive data is invalid.

Error 1181: $GROUP nesting too deep
Group names can be nested up to 10 deep.

Error 1182: $ENDGROUP without $GROUP
Unexpected $ENDGROUP, it needs an associated $GROUP directive.

Error 1183: Missing $ENDGROUP
Every $GROUP directive must have an associated $ENDGROUP.

Error 1184: Missing $GROUP name
A $GROUP directive does not have a group name.

Error 1185: Invalid $GROUP name
Group names must be valid symbol names.

Error 1186: Symbol too long, max. length is 80 characters
This probably means that the symbol name is longer than 80 characters. This includes the group

name.

Error 1187: Group names can't be PUBLic
Only the names of symbols defined within a group can be made public. The group name itself is not a symbol.

Error 1188: VOL attribute is for Flags only
Only EQUates of flags can have the VOL attribute because only Flags can be volatile (set to zero on reset or start-up). All other data types are non-volatile (in battery-backed memory).

Error 1189: RAM attribute is for Texts or DBs only
Only EQUates of Texts and DBs can have the RAM attribute because only Texts and DBs can be in Extension Memory RAM.

Error 1190: $AUTO numbers for VOL F must be before F
When defining the $AUTO ranges for Flags, the VOL (volatile) segment must be before the nonvolatile segment.

Error 1191: Text/DB numbers in extension memory must be >= 4000
When defining the $AUTO ranges for texts and DBs in extension memory the range must be between 4000 and 7999. Texts and DBs 0..3999 are not in extension memory.

Error 1192: Invalid array size
The array size in square brackets `[array_size]` is invalid. The start address + array size must not be greater than the last valid address.

Error 1193: Illegal type for array
The data type cannot be an array. Arrays of code blocks and constants are illegal.

Error 1194: User aborted
The user pressed the "Cancel" button which aborted the make.

Error 1195: Invalid $LIB file name: *filename*
The filename in a $LIB statement is not a valid filename.

Error 1196: Can't open $LIB file: *filename*
File filename cannot be found. The assembler searches first in the project's directory, then in the FBOX directory.

Error 1197: EXTN's type not the same as EQUate's
If the same symbol has both EXTN and EQU declarations in the same file, then the types must be the same in both. For example, this will generate the error because the types are different:
```
    EXTN MySym F     ;EXTN's type (F) not the same
    MySym EQU R 10   ;as the EQU's (R)
```

Error 1198: EXTNs have different types
If a symbol has more than one EXTN declaration in the same file, then the types of each declaration must be the same:
```
    EXTN Sym1 R    ;EXTNs have different types
    EXTN Sym1 T
```

Error 1199: Special operators not allowed in Texts or DBs
Special operators, or symbols derived from them, cannot be used to assign values in texts or data blocks. They can only be used for instruction operands, such as LD.

Error 1200: Pass 2 phase error

The value of a label or symbol on the second pass of the assembler is not the same as the value assigned to it on the first pass. This is often caused by incorrect forward references, or if a macro reference precedes the macro definition in the source file (macros must be defined before they can be called).

Error 1201: $AUTO for Text/DB cannot be above 3999 (4000..7999 are for RAM Texts/DBs)
The automatic address allocation range for Texts and DBs in the text segment must be between 0 and 3999, 4000 to 7999 are in extension memory and are for RAM Texts/DBs.

Error 1202: Macro parameter too long (max. length is 300 chars)
Macro parameter texts cannot be more than 300 characters long.

Error 1203: Unexpected $ENDCSF
Error 1204: Missing $ENDCSF
Error 1205: Invalid or missing hex data in $CSF segment
Error 1206: Bad or missing $CSF symbols
Errors which can occur in $CSF..$ENDCSF directives, see internal document DD-EPFW6-136 for $CSF directive details.

Error 1207: $USE not allowed in same file as $IFUSED
Error 1207: $IFUSED not allowed in same file as $USE
$USE and $IFUSED directives must be in separate files. $USE causes the file containing the associated $IFUSED directives to be assembled and linked.

Error 1208: Files containing $IFUSED cannot be assembled and linked
$USE causes the file containing the associated $IFUSED directives to be assembled and linked, so the file must not be in the 'Program Files' list, or must have it's property set to 'not Linked'.

Error 1209: Initialization value not allowed for this data type
Only Registers, Counters and Flags can have an initialization value.

Error 1210: Invalid initialization data value
Error 1211: Too many initializers
Error 1212: Missing initializers
Check the initialization data value(s) after the '='. If initializing and array, you cannot have more initializers than the size of the array (but you can have fewer).

Error 1213: Start-up initialization data handling is not implemented
Only 'first-time initialization' is implemented, which uses ':='. Start-up initialization is for a future release. For now, please initialise start-up values using IL code in XOB 16.

Error 1214: $DBXSEG not allowed inside a block
Error 1215: $DBXSEG not allowed inside $xxxSEG
The $DBXSEG directive cannot be used in this context.

Error 1216: Label has same name as symbol
Labels and symbols must have different names. For example, this will not work:
```
    Symbol  EQU 123
    ...
```
Symbol: INC Symbol ;label has same name as symbol
```
    JR  Symbol
```

Error 1217: Dynamic addresses cannot be used in global include files
Tip: Turn off the 'Use local declaration' advanced option in Symbol Editor
Symbols in global symbol files with dynamic addresses cannot be declared with EQU, because they would be assigned a different address in each file. If this file was created with the Symbol Editor, un-

check the symbol's "Advanced > Use Local Declaration" option on Symbol Editor's context menu. If the file was edited manually, either use an absolute address, or change the "symbol EQU .." statement to "EXTN symbol" and declare the symbol as public in another file using "symbol PEQU …"

Error 1218: $CSF data too long, max 512 hex characters per line (256 bytes)
A line of hex data in the $CSF..$ENDCSF segment can be max. 256 bytes long. That's 512 characters because there are two hex digits for every byte.

Error 1219: $LIB file has same name as a source file, please rename this file
Error 1220: $USE file has same name as a source file, please rename this file
All source files must have different file names, because the object files created are all in the same directory, and so must have different file names.

Error 1221: Text concatenation, <text> cannot insert external text
Error 1222: DB concatenation, <db> cannot insert external DB
To create a single Text or DB from several parts, all parts of the Text or DB must be defined in the same file.

Error 1223: Wrong $ENDGROUP name
The name used with $ENDGROUP is not the same as the opening $GROUP name, for example:

```
    $GROUP TheBeatles
    ...
    $ENDGROUP TheWho
```

Error 1230: Too many $ATTR directives, max is 128
A single symbol can have up to 128 associated $ATTR directives.

Error 1231: Expected symbol declaration EQU/PEQU/EXTN/DOC after $ATTR
$ATTR must be followed by a symbol definition because it assigns an attribute name to a symbol.

Error 1232: Duplicate $ATTR attribute
The same attribute name appears more than once.

Error 1236: TEQU temporary data symbol outside block
Temporary data can be defined only inside a block, see TEQU.

Error 1237: Invalid type for TEQU temporary data (use R or F)
Temporary data can be only Registers of Flags.

Error 1238: Multi-defined TEQU symbol
More than one temporary data item has the same name.

Error 1239: Illegal use of non-TEQU value
Attempt to assign a TEQU symbol using a non-TEQU symbol in the expression.

Error 1240: Illegal use of temporary address
Temporary data cannot be used in this context.
For example, you cannot use a temporary data address or symbol to declare a non-temporary symbol.

```
    Sym1 TEQU R [2]
    Sym2 EQU Sym1+0
```

You cannot load the address of a temporary register into another register, because register-indirect instructions cannot access it. They would interpret the contents of the register as a normal register address, not a temporary register address:

```
    LD   R 100
```

```
        Sym1     ;Error 1240
```

Error 1241: Invalid peripheral ID
The peripheral ID in an RDP or WRP instruction is invalid.

Error 1242: Invalid parameter type for OUT or INOUT
In $FBPARAM or $SFPARAM, the data type is not valid as an output parameter, e.g. constants cannot be written to.

Error 1243: Multi-defined FB parameter number
Error 1244: Too many FB parameters, max is 255
Error 1245: FB parameter name already used
Error 1246: PARAM declarations not supported
Error 1247: Missing $ENDFBPARAM
Error 1248: $ENDFBPARAM without $FBPARAM
Error 1249: Expected FB parameter declaration, missing $ENDFBPARAM?
Error 1250: Wrong $ENDFBPARAM name
Error 1251: FB already has $FBPARAM declaration
Error 1252: SF library function already has $SFPARAM declaration
Problem with $FBPARAM or $SFPARAM declaration.

Error 1253: Too many SF parameters
When calling a System Function with CSF, too many parameters have been used, see the library help.

Error 1254: Missing SF parameter declarations
Error 1255: $ENDSFPARAM without $SFPARAM
Problem with $FBPARAM or $SFPARAM declaration.

Error 1256: Invalid SF parameter
The parameter type used in the CSF call does not match the parameter type from the $SFPARAM declaration.
Note that some parameters must be a K type (e.g. untyped), and you will see this error if has a type. In this case, prefix the symbol with **K**, as in this example where `MyDB` is a DB type, but R or K is required::

```
MyDB EQU DB 4000
...
CSF    S.SF.DBLIB.Library  ;Library number
       S.SF.DBLIB.InitDBItems ;Initialise DB items with a Register value
       K MyDB              ;1 R|K IN, DB number (any DB number)
       K 0                 ;2 R|K IN, First DB item
       K 1                 ;3 R|K IN, Last DB item
       temp                ;4 R IN, Value to be written to all items
```

Error 1257: Missing FB parameter declaration
Problem with $FBPARAM or $SFPARAM declaration.

Error 1258: DEFTMP value too small
A temporary address greater than the DEFTMP array size has been used.

Error 1259: DEFTMP M only allowed inside COB or XOB 16
DEFTMP M is used to define the total amount of memory available for temporary data in each COB or all XOBs.
It is allowed only inside a COB or XOB 16.

Error 1260: Missing $ENDFUP

Error 1261: $ENDFUP without $FUP
$FUP..$ENDFUP are directives used by the Graftec editor S-Graf to delimit Fupla code inside a Graftce file (.sfc).
This error can only occur if S-Graf is generating invalid code. Please contact Saia Burgess Controls Technical support.

Error 1262: Invalid or missing $WRFILE file name, must be in double quotes
The file name in a $WRFILE statement must be enclosed in double quotes, e.g. `"filename.txt"`.

Error 1263: Cannot use LEQU or LDEF for system symbols
System symbols, with a single-character group name, e.g. `S.` or `A.`, cannot be local to a block or macro.

Error 1263: Too many $WRFILE files open, max. is 64
S-Asm can open a maximum of 64 different $WRFILEs in a single source file - but we recommend that you never use so many!

Error 1264: Block-local symbols cannot be blocks
LEQU cannot be used to define a symbol with a bock type, COB PB FB SB ST TR.

Error 1264: Can't open $WRFILE: *filename*
The file could not be opened, maybe the directory does not exist, the file exists but is write-protected or inaccessible.

Error 1265: Illegal use of external symbol or dynamic address in $WRFILE
$WRFILE creates the output at assembly time, so all symbols which it references must be defined at assembly time and not resolved by the linker.

Error 1265: Write error on $WRFILE: *filename*
Could be caused by a full disk or if another application has tried to access the file.

Error 1266: Cannot use $ATTR with DEF, LEQU or TEQU symbols
Attributes cannot be assigned to local, temporary or DEFined symbols, see $ATTR.

Error 1267: Unexpected $HIDE
Error 1268: Missing $ENDHIDE
$HIDE..$ENDHIDE directives are used by the IL Editor (S-Edit) to delimit the symbol definitions in a ".src" file which are inserted by Symbol Editor. When S-Edit opens the file, it removes the $HIDE..$ENDHIDE section and passes it to the Symbol Editor. If this error occurs it means that S-Edit or the Symbol Editor is generating invalid code, contact Saia Burgess Controls Technical Support.

Error 1269: GEQU or GDEF outside macro
GEQU and GDEF are only for declaring symbols inside macros.

Error 1272: Invalid offset
This probably means that the offset is invalid.

Error 1273: Instruction not supported
The instruction is not supported by this version of S-Asm.

Error 1274: Floating point not supported in expressions
Expressions cannot contain floating point numbers because the assembler's expression evaluator only supports 32-bit signed integers.
But you can use FLOAT and IEEE values in these conditional expressions: > ==, e.g. $IF symbol = 1.2.

Error 1275: Public symbols cannot be derived from Externals
This code is not allowed:
```
    EXTN Symbol1         ;external symbol, value unknown
    Symbol2 EQU Symbol1  ;Symbol2 is derived from external Symbol1
    PUBL Symbol2         ;it cannot be made public
```
To make a symbol public, the value of the symbol must be known at assembly time, and if it is derived from an external symbol then its value is not known. If EQU is used to define the base symbol, then it works:
```
    Symbol1 EQU R        ;value known (can be dynamic or absolute adds)
    Symbol2 EQU Symbol1  ;Symbol2 is derived from local Symbol1
    PUBL Symbol2         ;now it's ok
```
Tip: If `Symbol1` was defined in a global symbol file (.sy5), then use the Symbol Editor option "Advanced > Use Local Declaration" to change its definition in the global include file from EXTN to EQU, and use an absolute address.

Error 1276: Cannot use temp data (TEQU) in $INIT/$XOBEG/$COBSEG sections
Temporary data cannot be used in these segments if they are inside another block. But you can use temporary data in the XOB or COB itself, or if the $INIT/$XOBSEG/$COBSEG section is outside a block.

Error 1277: Multi-defined string name
The string name, defined with STR, has already been used. Strings defined with EQU can only be defined once. Use DEF if you want to re-defined the string with another value, or LEQU or LDEF to define a string which is local to the current macro or block.

Error 1278: Missing closing bracket ')'
A closing bracket is required, to match the opening bracket, for example in the @STR( ) and @ATTR( ) operators.

Error 1279: String name not defined
The string, referenced with @STR( ), has not been defined, see STR.

Error 1280: Invalid string name
String names must be valid symbol names, see STR.

Error 1281: Attribute not defined
The symbol does not have an attribute with this name, see @ATTR( ).

Error 1282: Strings (STR) cannot be declared with PEQU
Strings cannot be public, the string table is local to each file, see STR.

Error 1283: Value length exceeds specified width, e.g. symbol.04 where symbol is > 9999
The text contains a formatted symbol, and the symbol's value will not fit into the specified number of digits. This may happen for Mode C texts which are specified using this syntax: `"$",symbol.04T`, and the symbol is a register or flag number > 9999. For mode C texts which use more than 4 digits, firmware version >= 1.16.54 is required, and you must use the new extended format with an 'X' and a 5-digit address, e.g. `"$RX",symbol.05`.
Tip: If dynamic (auto-allocated) addresses are used, open the 'Build Options' and check the range for Registers or Flags does not go over 9999.

Error 1284: $DNLDFILE is not supported for Smart RIO programs
The program inside a smart RIO (PCD3.T665/T666) cannot contain additional configuration files. The RIO Manager can only download a single program file at this time.

Error 1285: Invalid $DNLDFILE file name (cannot contain spaces or accented chars, max.

length is 24)
File names in the PCD's internal file system have some restrictions:
- cannot be more then 24 characters long, including the extension and the dot
- cannot contain space, tab, or `\ / :`
- cannot contain characters with codes < 32 (space) or > 126, 7-bit ASCII only, no accented characters

Error 1286: More than one $IPADDS directive
The $IPADDS directive defines the IP address of the destination PCD. Only one IP address is allowed.

Error 1287: Macro not defined
The code looks like a macro call, but the macro definition has not been found. Maybe an include file is missing or the wrong version has been included (see $include search path), or the macro name is wrong. This error may be followed by 'Unexpected operand' errors for the macro parameters, because the macro call cannot be processed.

Error 1296: $ENDFOR without $FOR
$FOR loops must be closed with $ENDFOR. This error can also occur if there was a syntax error in the opening $FOR line.

Error 1297: Nested $FOR not supported
$FOR .. $ENDFOR loops cannot contain other $FOR loops.

Error 1298: $FOR symbol must be declared with DEF
The $FOR control symbol has already been defined, but it is not a DEFined symbol. You can re-use a DEFined symbol in $FOR, but not an EQU etc.

Error 299: $FOR start value > end value
$FOR can only increment values, so the start value must be less than the end value. But you can use another DEF symbol which you decrement in the loop, e.g.

```
    decsym DEF 1000
    $FOR incsym = 0 .. 9
        ...
    decsym DEF decsym - 1
    $ENDFOR
```

Error 1300: Missing $ENDFOR
The last $FOR has no associated $ENDFOR.

Error 1301: Macros cannot be called inside $FOR loop
Macro calls are not allowed inside $FOR. But $FOR can be used inside macros.

Error 1302: $FOR loop count exceeded, max. is 65536
The maximum loop count for $FOR is 65536. This prevents infinite loops.

Fatal Error Messages

Fatal Error 1300: No file name
The make file did not contain the names of any source file to be assembled.

Fatal Error 1301: Too many parameters
There are too many assembler switches in the make file, e.g. more than ten `/Dsymbol[=n]` switches.

Fatal Error 1302: Invalid switch "*switch*"
A command line switch in the make file is not a valid assembler switch.
For `/Dsymbol[=n]` switches, make sure the symbol name is not a reserved word and is not multi-defined.
The invalid switch has probably been defined in correctly on the "Build Options" dialog box in the "Additional build options" field.

Fatal Error 1303: Invalid file name: *filename*
A source file name is not a valid. Network paths are not supported, you must assign a drive letter.

Fatal Error 1304: Can't open file: *filename*
Fatal Error 1305: Read error on file: *filename*
Fatal Error 1306: Write error on file: *filename*
For reads: the source file does not exist. For writes: the disk or file is write protected, the disk is full, the file is open in another application, or you do not have the correct access rights.

Fatal Error 1308: Out of memory
This will normally never occur because the PC can use "virtual memory" (disk space) if it runs out of conventional memory. Try booting the PC, or hitting it with a hammer.

Fatal Error 1316: Program not licensed
The user key file does not allow program to be built. Obtain an updated license from your Saia Burgess Controls representative.

Fatal Error 1317: Recursive $INCLUDE file: *filename*
An include file includes itself. Note that it may not directly include itself, but may be included by a file which it includes.

Fatal Error 1318: Line *n***: Stack overflow**
An expression is too long or too complex to be evaluated.

Fatal Error 1320: *user_defined_message*
This fatal error is generated by the $FATAL directive in the user program.

Fatal Error 1321: FBD code in SFC module not compiled
If a Graftec SFC module contains Fupla FBD code in one or more of its steps or transitions, this error is generated if the Fupla code has not been compiled.

Fatal Error 1322: $LIB file has same name as source file, please rename your file: *filename*
A source file in the user program cannot have the same name as a Fupla FBox library file. Change the name of source file filename. This can be done from the Project Manager's 'File Properties' dialog box, which also renames the file in the Project Tree (do not use Explorer to rename the file, because this will not rename the associated files too).

Fatal Internal Error: file: Line *line***:** *reference text*
The assembler detected an internal error. The reference text indicates the location. Notify Saia Burgess Controls Technical support, and provide the *reference text* and details of how to reproduce the error.

20.2 Assembler Warnings 1500+

Warnings do not stop assembly and the object file is still created.

Warning 1502: Parameter *name* **not used in macro** *name*
The formal parameter given in the macro definition is not referenced by the code in the macro body.

Warning 1503: Too many parameters for macro *name*
More parameters have been supplied with the macro call than are used by the macro.

Warning 1504: Step or Transition has no output
For a complete Graftec structure, each ST and TR should have an output so that processing can continue.

Warning 1505: SB has no Initial Step (IST)
The initial step is the entry point of the SB.

Warning 1506: *user_defined_warning_message*
This warning is generated by the $WARNING directive

Warning 1507: Invalid expression or unresolved symbol between @..@ in $directive
The expression between '@' characters in a $TITLE, $STITLE, $REPORT, $ERROR, $ONERROR or $WARNING directives is not valid. To place an @ character in the text, use @@. See Using symbols in $directives.

Warning 1508: More than one Initial Step (IST) in SB
Each sequential block should contain only one initial step.

Warning 1509: RCOB instruction only allowed in COB or XOB
The RCOB instruction should only be used in a COB or XOB.

Warning 1510: DEFVM operand not same as /VOLF=address
The DEFVM instruction has been used in an old Fupla file, and it does not match the new Volatile Flags settings defined in Project Manager's "Build Options".

Warning 1511: Bad ';; *item***:' format: Fupla file needs re-compile**
The Fupla compiler file being assembled was created with an early version of S-Fup, and it does not contain the correct format of cross-reference information. This means that the cross-reference list window will not show the Fupla block and page for a symbol. The module should be re-compiled (do a 'Rebuild All').

Warning 1512: Local and global symbol (PUBL/EXTN) with same name: *symbol*
A local symbol with the same name as a global symbol will be used instead of the global. This may be a programming error. To be safe, do not give local and global symbols the same name.

Warning 1513: Symbols '*symbol1***' and '***symbol2***' have same type/value**
Warnings can be issued after a successful build if two or more symbols are defined with the same type and value. This can help find obscure problems in your code. This warning can be turned off from the 'Build Options', after you have made sure that the warnings are not programming errors.

These could be serious programming errors, or they may be intentional. For example, if two symbols address the same Register then it may mean that the Register is being illegally modified by another part of the program. But if it is a workspace Register, then it can be re-used elsewhere without problems, so you may want to turn off this feature.

A common programming practice is to define a base address symbol, then define offsets from this symbol, for example:

```
    BaseFlag EQU F 100
    Flag1    EQU BaseFlag+0
    Flag2    EQU BaseFlag+1
```

In this case there would be a warning because BaseFlag and Flag1 have the same type and value (F 100), but this is not a programming error. Some SAIA PG5 libraries use this technique, and so warnings may be issued. Fupla programs use this technique a lot, so these warnings have been suppressed.

Sometimes the offset may be used in the operand, for example:

```
    STH BaseFlag+1    ;same address as Flag2!!!
```

This is a programming error, symbol Flag2 should be used. SAsm will warn in this case too, and the lines can be found in the Cross-reference List.

Note
If the BaseFlag symbol is Global (defined as EXTN BaseFlag), then the warning will be issued, but the references to its absolute address will not be in the Cross-reference List, because this error is found by the linker, not the assembler.
Tip: To find out where the errors are, use Project Manager's 'Data List View' and sort the symbols by Type. Find the absolute address, e.g. "F 1000", and use the 'Cross-reference List' command to see where the address is used.

Warning 1514: Step or Transition outside Sequential Block
STs and TRs are normally defined inside the SB..ESB block. STs and TRs outside any block are allowed, but S-Graf will NOT be able to open them. Files with STs and TRs outside the SB can only be opened by SEdit. STs and TRs will be outside the SB if they were downloaded as 'changed blocks' into the PCD, and the program was uploaded and disassembled. Try to move the STs and TRs back into the correct SB.
Note: The IST (Initial Step) must be inside the SB. STs and TRs cannot be inside any other block.

Warning 1515: Dangerous expression for FB parameter number
FB parameter numbers are hard-wired constants. This warning is issued when an expression is used for an FB parameter number, which usually means that the wrong parameter will be referenced. For example:

```
    FB   MyFB
    Param1 DEF = 1     ;parameter 1
    Param2 DEF = 2     ;parameter 2
    LDL  R 0
         Param1+1      ;wrong! references parameter 2
```

This is sometimes done when the user wants to increment the value of the parameter, not the parameter number itself.

Warning 1516: Symbol is not an array: *symbol_name*
A common programming error is to reference data using an offset to a symbol which is not an array, for example:

```
    Symbol EQU R 100    ;single register
    Array  EQU R 101[2] ;array of registers
    INC    Symbol+1     ;wrong! increments R 101 (Array[0])
    INC    Array+1      ;OK, increments R 102 (Array[1])
```

But there is one programming case which is valid and may be intentional, but will still cause this warning:

```
    Array1 EQU R [10]   ;array of 10 registers
    Array2 EQU Array1+5 ;could be array of 5 registers, or 1 register
    INC    Array2+1     ;warning: symbol is not an array
```

In this case Array2 is assigned as a slot in Array1, and the S-Asm assumes it is a single register, not an array, so it issues the warning.
Tip: This warning can be turned off from the "Build Options".

Warning 1517: Library not found: *lib_name, version, distributor: path*
A library was checked in the Library Manager, but the library could not be found. If the library is used

in the program, then there will be build errors. If the library is not used then this warning can be ignored, or you can un-check the library in Library Manager.

Warning 1521: *filename***: Line 65536: More than 65535 lines, cross-reference list will be invalid**
A single source file can now contain any number of lines - but the cross-reference list line number is limited to 65535 (16 bits). If the source file has more than 65535 lines, then the line numbers for the Cross-reference List will be wrong, they will roll over from 65535 to 0. This warning can sometimes be issued for very large Fupla files. If possible, break the file up into two or more smaller files. Note that each $include file can also contain up to 65535 lines.

Warning 1522: $CPU is now obsolete
PCDs do not contain more than one CPU anymore, so this directive is no longer needed. It is ignored.

Warning 1523: Ignoring AND after XOR instruction
To follow XOR with an ANH/ANL instruction, first end the sequence with an OUT to store the XOR result, then start a new linkage with STH/STL. An AND linkage after XOR will be ignored.
For example:

```
    STH   I 0
    XOR   I 1
    ANH   I 2   ;warning 1523
    OUT   O 16
```

The `ANH` instruction is ignored after the `XOR`, so the above code is equivalent to:

```
    STH   I 0
    XOR   I 1
    OUT   O 16
```

The correct IL code should be:

```
    STH   I 0
    XOR   I 1
    OUT   F 123 ;temp flag
    STH   F 123 ;start new linkage
    ANH   I 2
    OUT   O 16
```

Warning 1524: $IFB/$IFNB only processes the first parameter in < >, the others are ignored
Older versions of S-Asm accepted more than one < > parameter on the same line, but only the first parameter was processed as party of the $IF expression, the others were ignored. $IFB was sometimes used to reference unused macro parameters to prevent the "unused macro parameter" warnings. But now we give a warning if there is more than one < > on the line, because this may be a programming error if the programmer intended them to be part of the $IFB/$IFNB expression.

20.3 Linker Errors 2000+

When an external reference is resolved using the associated public symbol, the range of the value produced and its type are checked. Errors occur if the symbol cannot be resolved, the value is out of range or the data types are not compatible.

Format
Linker error messages have the form:
Error *n***:** *file* **(***offset***):** *symbol***:** *description*

Where:

n	The error number, see the messages below.
file	The name of the object file containing the error.
offset	The relative offset into the code segment of the error. This matches the relative line numbers in the listing file.

	Note: The number may be one or two lines after the actual error if the operand is dependent on the value of a preceding operand.
symbol	The name of the external symbol which caused the error.
description	See messages below.

Linker Error Messages
For simplicity, the file, offset and symbol name are not shown.

Fatal Error 2000: Missing file names
No object file name is available. Invalid make file?

Fatal Error 2001: Too many file names
A maximum of 256 object files can be linked to form one PCD's user program.

Fatal Error 2002: Invalid switch
The make file contains an invalid linker switch.

Fatal Error 2003: Invalid file name: *filename*
The make file contains an invalid file name or path. Network paths are not supported, you must assign a drive letter.

Fatal Error 2004: File specified more than once: *filename*
The input file name is the same as the output file name, or the same object or library file has been specified more than once.

Fatal Error 2005: Unexpected "="
Error in the make file. Only one "=" can appear on the command line or in the command file.

Fatal Error 2006: Can't open file: *filename*
The specified file cannot be found, or cannot be created.

Fatal Error 2007: Read error on file: *filename*
Fatal Error 2008: Write error on file: *filename*
The specified file cannot be read or written. The file may not exist, disk may be full, not ready, or an existing file may be read-only. Perhaps the file is already open in another application.

Fatal Error 2009: Out of memory
Should never happen. The PC does not contain enough available memory to complete the linkage. Try booting the PC.

Fatal Error 2012: Invalid object file: *filename*
Fatal Error 2013: Unexpected end of file: *filename*
The object file is not an object file produced by the Saia PG5 assembler, or may have been created with a newer version of the assembler.

Fatal Error 2014: More than 100 errors
Linkage is aborted if there are more than 100 errors.

Fatal Error 2017: Bad object file version: *filename*
The object file was created using a very old version of the Saia PG5 assembler and is not compatible with this version of the Build Utility. Re-assemble the source file. If it is a ".obl" file then a 'Rebuild All' is needed.

Error 2020: Multi-defined public symbol: *symbol***, also in file(s):** *file_list*
The same public symbol has been declared PUBL or with PEQU in more than one file.

Error 2021: Unresolved external symbol: *symbol*
An external symbol, declared with EXTN, does not have an associated PUBLic or PEQU symbol definition. Or a symbol which is derived from an External symbol is made Public and the linker cannot resolve its value, for example:
File1.src
```
    EXTN ExtnSymbol
    PublSymbol EQU ExtnSymbol
    PUBL PublSymbol
    ...
```
File2.src
```
    EXTN PublSymbol
    ...
    INC PublSymbol    ;causes Error 2021
    ...
```
If the Public symbol is defined in a global include file which is edited by Symbol Editor, e.g. Global. sy5, then the solution is to set the context menu's **Advanced > Use Local Declaration** option for the symbol. This causes it to be defined with EQU in the include file instead of EXTN. This problem has been seen with Heavac symbols, A.HVC.xxx etc, and is solved with the Use Local Declaration option.

Error 2022: Multi-defined instruction: *instruction***, also in file(s):** *file_list*
The following instructions can appear only once in a program, multi-defined instruction errors occur if more than one is found:

`DEFTB`	Define timebase.
`DEFTC`	Define timer/counter partition.
`DEFVM`	Define volatile flag memory.
`DEFTR`	Define timer resolution.

Error 2023: Multi-defined block: *block***, also in file(s):** *file_list*
Each COB, XOB, PB, FB, SB, IST, ST, TR, TEXT or DB (Data Block) can be defined only once in a program for a single PCD. For example, there can be only one COB 0 in a program. Multi-defined block errors occur if there is more than one definition of the same block.
Notes
- Data Blocks and Texts share the same numbering. For example, if DB 10 exists, then TEXT 10 cannot be defined, and vice versa. This will cause a multi-defined Text or DB error according to which type was processed first.
- DB 3999: If the use of dynamic addressing is turned off (see Project Manager's "Build Option"), then DB 3999 is reserved by the PG5 to hold the Unique Program Identifier, and DB 3999 cannot be used by the user the program.

Error 2024: Undefined block: *block*
Each COB, XOB, PB, FB, SB, IST, ST, TR, TEXT or DB reference must have an associated definition, undefined clock errors occur if it doesn't. For example, if "CFB 1" (call Function Block 1) appears in the program, then "FB 1 ... EFB" (Function Block 1 definition) must also be present.

Error 2025: File: *file* **Line** *n***: Invalid FB parameter: CFB** *fb***: Param** *p***. See file** *fbfile***: Line** *x*
Parameter *p* of Function Block *fb* does not match the parameter expected by the Function Block definition in file *fbfile* on line *x*. This error message contains the source file line number, so double-clicking on the error message will open the source file at the correct line.

Error 2026: File: *file* **Line** *n***: Missing FB parameters: CFB** *fb*
The Function Block call contains too few parameters. This error message contains the source file line number, so double-clicking on the error message will open the source module at the correct line.

Error 2027: Invalid character in text

The value of an externally defined character in a Text is 0 (reserved for NUL) or is greater than 255.

Error 2028: Invalid SASI text
Texts defined inside $SASI..$ENDSASI directives are invalid after resolving externals and creating the actual text.

Error 2029: Text too long
Texts 0..3999 can contain up to 3072 characters. Texts 4000+ can contain up to 65535 characters. This error occurs if the text is longer than the maximum length.
If the text is > 3072 characters, and a dynamic text number has been used, check that text is not being assigned an address below 4000. The text numbers for dynamic addresses are assigned from the Build Options. Either change the range, or use an absolute text number >= 4000.

Error 2030: Illegal use of typed external
The external has been given a type (I, O, F etc) and can't be used in this context.

Error 2031: Invalid memory flag
Accepts 0 or 1.

Error 2032: Invalid input/output number
The maximum number of I/Os is dependent on the PCD model, refer to your hardware documentation.

Error 2033: Invalid flag number
Old PCD models support Flags 0..8191. From firmware version 1.14.00 this was increased to 0..14335.
PCDs with firmware version 1.20.00 support 0..16383 flags, but this must be enabled from the Build Options "Use 16-bit addressing".

Error 2034: Invalid timer/counter number
The Timer/Counter partition can be manually selected from Project Manager's "Build Options", see the 'Last Timer' setting. Timer/Counter addresses are 0..1599.

Error 2035: Invalid register number
Old PCD models support registers 0..4095. The newer "NT" systems (PCD3 etc) support 0..16383. For firmware versions before 1.20.00, the Register Indirect instructions (CPBI, SASII, SCONI, SRXMI, STXMI, TFRI) cannot use Register addresses above 8191. If the error is from a symbol with an absolute address, change it to use an address <= 8191. If it's a dynamic address, open the Build Options and change the dynamic address range for Registers to start below 8191 and do a "Rebuild All". If the register's address is still assigned above 8191, try changing the link order of the files which fail, putting these at the beginning, and do a "Clean Files" operation to re-assign the dynamic addresses.
Firmware versions 1.20.00 and above allow the full range of Register addresses 0..16383 in all Register Indirect instructions - but this must be enabled from the Build Options "Use 16-bit addressing" setting. This creates a PCD file containing code in a new format, which is not compatible with older PCD firmware.

Error 2036: Invalid constant
A "K" constant is a 14-bit value stored in the PCD's 16-bit operand. The upper 2 bits of the operand define the data type. A "K" constant is used only in operands which require a data type (R/F/T/C/I/O/K etc). These errors can occur for K constants which or not in the range 0..16383 (0..3FFFH).
The LDL and LDH instructions load 16-bit values, 0..65535.
Other constants and symbol values, and values for the LD instruction, are 32 bit values, range -2147483640 . .2147483647.

Error 2037: Invalid type
The external's type (I, O, F, COB etc) is invalid. The external symbol cannot be used in this context.

Error 2038: Incompatible data types
The operand or public symbol already has a type, and it is not the same as the external declaration. Or the operand types of the instruction are not compatible.
For example, File1 contains a public symbol definition:
```
    Symbol PEQU F
```
File2 contains and external symbol reference:
```
    EXTN Symbol F
```
If the type of the public symbol is now changed to `I` (for example), Error 2038 will occur because the type of the `EXTN` reference is still `F`, and the expected and actual types are different.
If the external reference is in the Symbol Editor of a Fupla or IL file, you can update the external declaration simply by opening and saving the file (the correct type is taken from the public symbol table created by the build).

Error 2039: Invalid priority
Error 2040: Invalid PB number
Error 2041: Invalid SB number
Error 2042: Invalid FB number
Error 2043: Invalid COB number
Error 2044: Invalid XOB number
Error 2045: Invalid ST/TR number
Error 2046: Invalid test number
Error 2047: Invalid nibble number
Error 2048: Invalid bit number
Error 2049: Invalid byte number
Error 2050: Invalid word number
Error 2051: Invalid long word number
Error 2052: Invalid digit number
Error 2053: Invalid number of elements
Error 2054: Invalid timebase
Error 2055: Invalid semaphore
Error 2056: Invalid exponent
Error 2057: Invalid text number
Error 2058: Invalid serial channel
Error 2059: Invalid switch
Error 2060: Invalid station number
Error 2061: Invalid control signal
Error 2062: Invalid data block number
Error 2063: Invalid counter channel
Error 2064: Operand must be zero
The operand is out of range or the number is invalid. See Data Types for valid ranges. Some ranges are dependent on the PCD type or FW version.
Some instructions do not allow the full range of element addresses or values, and the range of some operands is dependent on the values of preceding operands (SRXM, STXM, TFR, BITI, BITO, DIGI, DIGO etc).

Error 2065: Register indirect, not a register
For register indirect addressing (TFRI/STXMI/SRXMI etc. instructions), the symbol must be a Register.

Error 2066: DEFTR and DEFTB in same program
Only one timebase mechanism can be used in the same program.

Error 2067: Different $AUTO declarations in file: *filename*
The $AUTO directive is generated by Project Manager from the dynamic address ranges in the "Build Options".

$AUTO is used to define the dynamic address space of dynamic address allocation. Normally you should have only one set of $AUTO declarations in one source file.

Error 2068: Dynamic address overflow for type R: Has *nnn* (*nnn..nnn*), uses *nnn*, needs *nnn* more / needs "Clean Files" operation
The program uses more dynamic addresses than have been defined in the Build Options.
For F, R, T, C, TEXT and DB types, the dynamic range is defined by Project Manager's "Build Options", so the range must be increased there.
For block types (COB, PB, FB, ST, TR and SB) the range is fixed, so this error means that too many blocks of that type have been used and the program must be re-coded to reduce the number of blocks.
If using arrays, it is possible that there are not enough consecutive addresses available to hold the array, even if there are enough free addresses. In this case a "Clean Files" operation must be done, to re-allocate all the addresses.

Error 2069: Absolute address in dynamic address segment: *type adds*
An absolute address (e.g. R 100) is used which is within the dynamic address range defined in Project Manager's "Build Options". This would overwrite data used somewhere else, so it's not allowed.
For new PG5 programs you should find the address and change it to remove the conflict - use Project Manager's "Data List" view or "Find in Files" feature to help you find where the address is used.
In some communications FBoxes, the absolute address may be in a remote station but S-Asm does not know this, so you must choose a different absolute address in the remote station (the absolute address must NOT be in the dynamic address space of the local or the remote station).
If you have imported a PG4 program then you should open the Build Options and modify the dynamic address ranges or the absolute addresses so they do not conflict.
Tip: The **IPSend FBox** can generate this error if you use an absolute address for the remote station's destination. Change it to an address which does not conflict with the dynamic address ranges of the local and remote stations.

Error 2070: Different $CPU number in file: *filename*
If more than one source file contains a $CPU directive, the CPU numbers must all be the same.
Note: $CPU is now obsolete, new PCDs can have only one CPU.

Error 2071: Different $STATION number in file: *filename*
If more than one source module contains a $STATION directive, the station numbers must all be the same.

Error 2072: Too many $PCDVER directives
A maximum of 20 different $PCDVER directives can be used in one program.

Error 2073: Missing DEFTC instruction
The DEFTC instruction is normally generated by Project Manager from the "Build Options", so this error will normally never occur.
If dynamic resource allocation is used for Timers or Counters, then the program must contain a DEFTC instruction to define the number of Timers.

Error 2074: DEFTC incompatible with $AUTO T or $AUTO C
The DEFTC instruction and $AUTO declarations are generated by Project Manager from the "Build Options", so this error should never occur.
If $AUTO declarations for Timers or Counters are used, the DEFTC instruction (DEFine Timers/ Counters) must be compatible with the $AUTO values.

Error 2075: Missing DEFVM instruction
The DEFVM instruction is normally generated by Project Manager from the "Build Options", so this error will normally never occur.

If separate dynamic allocation is used for volatile flags ($AUTO VOL F), then a DEFVM instruction must be used to define the volatile flags. Without a DEFVM instruction all flags are non-volatile by default.

Error 2076: DEFVM incompatible with $AUTO VOL F or $AUTO F
The DEFVM instruction is normally generated by Project Manager from the "Build Options", so this error will normally never occur.
If $AUTO declarations for flags are used, the DEFVM instruction (DEFine Volatile Memory) must be compatible with the $AUTO values.

Error 2077: No $AUTO declaration for type: *type*
The $AUTO directive is generated by Project Manager from the dynamic address ranges in the "Build Options". Dynamic address allocation cannot be used for type unless it has a $AUTO declaration.
This can occur for the following reasons:
- "Dynamic addressing enabled" has been set to "No" from Project Manager's Build Options, but the program uses dynamic address allocation.
- The Device Configurator has "IP Transfer Protocols - Initialize Open data Mode" set to Yes, and the Build Option "Dynamic Addressing Enabled" is No.
- "Has Volatile Flags" is set to "No" in the Build Options, and the program uses a volatile Flag (F VOL).

Error 2078: No COB present
At least one Cyclic Organization Block, usually COB 0, must be present in the program. This error occurs if there are no COBs.

Error 2079: Array bounds overflow
An offset was added to a symbol which is defined as an array, and the resulting address is outside the array. For example:
File1.src
```
    ArraySym EQU R 100[10]   ;defines registers 100..109
    PUBL ArraySym
```
File2.src
```
    EXTN ArraySym
    Item11 EQU ArraySym+10   ;References Register 110, array bounds overflow
```

Error 2080: Old object file version, please re-assemble: *file*
Object files created by very old versions of the Saia PG5 assembler cannot be linked. The source code must be re-assembled to produce an object file with the new format.

Error 2081: More that 383 elements in DB *n*
Data blocks 0..3999 can have a maximum length of 383 elements. When DB numbers are dynamically allocated, the check for the number of elements being greater than 383 is done after the DB number has been assigned by the linker.

Error 2082: Value in DBX *n* **too big:** *symbol*
The value of a DBX data item is larger than the number of bytes declared for the DBX item, e.g.
@3: 0FFFFFFFFh   ;needs 4 bytes not three

Error 2083: Invalid DBX number: *n*
DBXs are numbered 0..3999.

Error 2084: All COBs used, no COB number available for $COBSEG segment
When automatically allocating a COB number of the $COBSEG segment, all COB numbers have already been used.

Error 2085: All COBs used, no COB number available for SB calls

S-Asm can automatically create a COB to contain the CSB (Call Sequential Block) instructions for Graftec programs.
See Project Manager's Build Option "Generate code to call uncalled Graftec SBs. This error occurs if there are no more unused COB numbers.

Error 2086: JPI/RCOB has invalid address in *block*
If the $INIT, $COBSEG or $XOBSEG directive has been used to insert code at the start of a block, the jump offsets for JPI or RCOB instructions for that block will be incorrect. These instructions cannot be used in blocks which have an unknown amount code inserted at the beginning.

Error 2087: Multi-defined Unique Program Identifier DB name
If name is 3999: Data Block DB 3999 is reserved to hold the Unique Program Identifier, and cannot be used in your program, change your DB number to something else.
If name is __PCD_UID__: This means there is more than one 'unique program identifier' Data Block in the program, try a 'Rebuild All'.

Error 2088: Volatile Flag address not in Volatile Flag segment
The address of an F VOL flag was not in the Volatile Flag address space defined in the Build Options.

Error 2089: NUL character in Text *name*
Texts numbered 0..3999 (in Text memory) cannot contain the NUL character (00) because this is used as the end-of-text marker. Only Texts 4000 and above, in Extension Memory, can contain NUL characters.
Tip: If the Text number is dynamic (auto-allocated), open the 'Build Options' and change the dynamic address range for the Text or RAM Text to 4000 or above.

Error 2090: Multi-defined $CSF library function: *library_name*
The library function has been defined more than once in one of the $CSF..$ENDCSF sections.

Error 2091: Protected library used: *library_name*
The library is software protected. Please contact your Saia Burgess Controls representative or the library distributor. You can find the distributor using Project Manager's "Library Manager".

Error 2092: $LIB files have the same name: '*filename1***' and '***filename2***'**
Two $LIB statements define a file with the same name but it is not the same file. $LIB assembles and links the file, and it is not possible assemble and link two files with the same name (the OBJ files would have the same name). This error can occur if you have a source file with the same name as a library module, e.g. LONNET3, in this case you must rename your source file.

Error 2093: Execute failed: *description*
The build procedure has failed to find an application which is required by a library or add-on tool. The description will help determine what's wrong. Try re-installing the library, or contact the distributor.

Error 2094: Symbol is not an array
An offset added to a dynamically allocated symbol will form an address which may have been assigned to another dynamic symbol. The code will probably not run properly. To access offsets fro a dynamically allocated base address, the base address must be declared as an array. For example:

```
    RegBase EQU R      ;not an array
    INC RegBase+1      ;error! register used as other dynamic address
    ...
    RegBase EQU R[2]
    INC RegBase+1      ;ok, RegBase is an array of 2 registers
```

Note: It is not possible to create a sub-array from an array.
In the example below, Array2 will not be be an array, it will be a single Register address at offset 2, and Array3 will be a single Register address at offset 4 (2+2).

```
Array1 EQU R [10];
Array2 EQU Array1 [2]
Array3 EQU Array1+2 [2]
```

Error 2095: Local and public symbols have same name but different type/value
If a public and a local symbol have the same name and a different type or value then it is a fatal error because S-Asm does not know which one to use. If their types/values are the same then they are assumed to be the same symbol.

Error 2098: Invalid peripheral ID
The peripheral ID for an RDP or WRP instruction is invalid.

Error 2099: Failed to create Block Information files
This error can occur if the files in the '.\Sym' subdirectory are read-only, or if the existing block information files were created with an old pre-release $ version of the PG5. If the files may be old versions, use Project Manager's "Device / Advanced > Clean Files" command to delete the old block information files. The files will be re-created when the program is rebuilt and downloaded.
Tip: This error has also occurred if the same block is defined more than once in the user program, and S-Asm did not generate an error (this can sometimes happen if PUBL/EXTN and dynamic allocation is done in a certain order). Use Project Managers's 'Data List View' to display all the blocks, sort by type/number and check for duplicate block definitions.

Error 2101: Too many initialization values (>64K of data)
The total amount of first-time initialization data cannot exceed 64K bytes. Remove all unnecessary first-time initialization data, and/or write some code to initialize the values.

Error 2102: Internal error: *error_id*
The linker detected an internal error. Please contact Saia Burgess Controls Technical Support and report the *error_id* and how to reproduce the problem.

Error 2103: Object file error: *filename*: **Psn** *p*: *error_id*
The linker detected a problem in an object file.
Please contact Saia Technical Support and report the *error_id* and how to reproduce the problem.

Error 2104: *pcd_file_name*: *error_msg*
An error occurred while trying to create the 'Downloadable Files Information Database'. The build was successful, but it means that the 'Download Changed Files' option will not know which downloadable files have changed. Downloadable files are used for configuration data, such as the BACnet configuration.

Error 2105: Value length exceeds specified width, e.g. symbol.04 where symbol is > 9999
The text contains a formatted symbol, and the symbol's value will not fit into the specified number of digits. This may happen for Mode C texts which are specified using this syntax: `"$",symbol.04T`, and the symbol is a register or flag number > 9999. For mode C texts which use more than 4 digits, firmware version >= 1.16.54 is required, and you must use the new extended format with an 'X' and a 5-digit address, e.g. `"$RX",symbol.05`.
Tip: If dynamic (auto-allocated) addresses are used, open the 'Build Options' and check the range for Registers or Flags does not go over 9999.

20.4 Linker Warnings 2300+

Warnings do not prevent the creation of the ".pcd" file.

Warning 2301: *filename*: **Line** *n*: **FB parameter not referenced: FB** *n*: **Param** *n*
This occurs when a Function Block parameter, supplied in a "Call Function Block" (CFB) parameter list, is not used by the Function Block. The source module name and line number are shown, so

double-clicking on the error message will open the source file at the correct line.

Warning 2302: Can't sort symbol table
There is not enough memory available to sort the global symbols into alphabetical order. The symbols are listed in the map file unsorted. (obsolete)

Warning 2303: *filename*: **Line** *n*: **Too many FB parameters: CFB** *n*
The Function Block call provides more parameters than the FB actually needs. The source module name and line number are shown, so double-clicking on the error message will open the source module at the correct line.

Warning 2305: Failed to create debug info file: *filename*
For debugging with SEdit's 'code view' mode, the linker creates a file containing the source module line numbers and the locations of the actual code in the PCD's memory. These files are created in the 'Sym' subdirectory. This error occurs if the file could not be created. Check the 'Sym' directories files are not read-only, then try a 'Rebuild All'.

Warning 2306: More than one initialization value for: *type address symbol*
If type address is EQUated more than once, which is allowed, it could have more than 'first-time initialization' value. S-Asm will always use the last value supplied. This warning is issued even if the initialization values are the same.

Warning 2307: Local and public symbols have the same name: *symbol*
An EQUated symbol which has not been made PUBLic on one file has been defined in anotehr file and made PUBLic (or PEQU was used). This could be a programming error, or could be intentional, for example if the symbol was defined in an include file.

Warning 2308: Failed to copy PCX file (symbol database)
The ".pcx" file is always created even if the build fails. If the build was successful and there were no errors, the ".pcx" file is copied to the ".pcd" file. If there were errors, the ".pcd" file is deleted. The ".pcx" file contains the symbol table from the build which is read by the Symbol Editor.
This error can occur if the ".pcx" file is open when the build is done, and it cannot be updated.
If this error occurs, then you should close all applications which may be accessing the file (all editors and add-on tools), and do a manual build.

Warning 2309: Time-limited library used: *library_name* **: Expires:** *expiry_date*
The library has a limited license and will eventually expire. If the license expires then you will not be bale to build the project. Consider purchasing a license.

Warning 2310: Failed to create Block Information files
An error occurred while trying to create the 'Block Information Database'. The build was successful, but it means that the 'Download Changed Blocks' option will not know which blocks have changed, and this feature will be disabled.

Warning 2311: Failed to create Downloadable Files Information
An error occurred while trying to create the 'Downloadable Files Information Database'. The build was successful, but it means that the 'Download Changed Blocks' option will not know which downloadable files have changed. Downloadable files are used for configuration data, such as the BACnet configuration.

Warning 2312: Symbol is not an array: *symbol_name*
The symbol is referenced like an array, but it has not been defined as an array. This is often a programming error, and it should be checked. For dynamic addresses this is a fatal error. But for absolute addresses it is a warning because IL code written before arrays were introduced often used the "symbol+offset" syntax. You can disable this warning by setting the Build Option 'Warn on offset to symbol which is not an array' to 'No'.

```
EXTN Stmbol R   ;Symbol is R 100 (it is not an array, and not dynamic)
...
INC Symbol+1    ;Warning 2312: Symbol is not an array (R 101 is incremented)
```

Warning 2313: Symbols with same type and values
  R 123: Symbol 'symbol0', File: Untitled1.src
  R 123: Symbol 'symbol1', File: Untitled1.src
If the device's Build Option 'Warn on symbols with same type and value' is Yes, then S-Asm gives this warning whenever it finds two different symbols which are accessing the same address. This may be by design if an alias is used, or it may be a programming error. FBox libraries often use aliases for parameters, so you may see these warnings for Fupla programs - they can be ignored, and you can set the warning Build Option to 'No'.

21 Miscellaneous

21.1 ANSI and DOS Character Sets

ANSI character set
The ANSI character set shares the same characters with values 0..127 (00..7F hex) as the original DOS character set.
IL source files use the ANSI character set.

The following table shows Windows-1252, with differences from ISO-8859-1 outlined.
Each character is shown with its Unicode equivalent right below and its decimal code at the bottom.

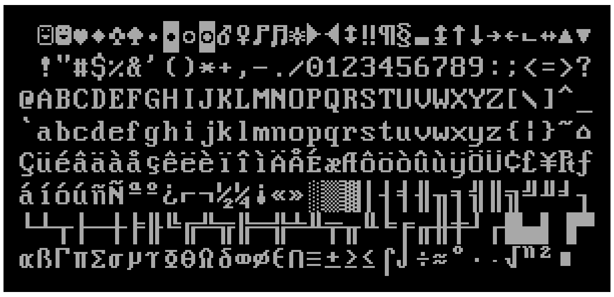
Legend: yellow cells are control characters, blue cells are punctuation, purple cells are numbers, green cells are ASCII letters, and tan cells are international letters.

Windows-1252 (CP1252)

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	NUL 0000 0	SOH 0001 1	STX 0002 2	ETX 0003 3	EOT 0004 4	ENQ 0005 5	ACK 0006 6	BEL 0007 7	BS 0008 8	HT 0009 9	LF 000A 10	VT 000B 11	FF 000C 12	CR 000D 13	SO 000E 14	SI 000F 15
1–	DLE 0010 16	DC1 0011 17	DC2 0012 18	DC3 0013 19	DC4 0014 20	NAK 0015 21	SYN 0016 22	ETB 0017 23	CAN 0018 24	EM 0019 25	SUB 001A 26	ESC 001B 27	FS 001C 28	GS 001D 29	RS 001E 30	US 001F 31
2–	SP 0020 32	! 0021 33	" 0022 34	# 0023 35	$ 0024 36	% 0025 37	& 0026 38	' 0027 39	(0028 40	) 0029 41	* 002A 42	+ 002B 43	, 002C 44	- 002D 45	. 002E 46	/ 002F 47
3–	0 0030 48	1 0031 49	2 0032 50	3 0033 51	4 0034 52	5 0035 53	6 0036 54	7 0037 55	8 0038 56	9 0039 57	: 003A 58	; 003B 59	< 003C 60	= 003D 61	> 003E 62	? 003F 63
4–	@ 0040 64	A 0041 65	B 0042 66	C 0043 67	D 0044 68	E 0045 69	F 0046 70	G 0047 71	H 0048 72	I 0049 73	J 004A 74	K 004B 75	L 004C 76	M 004D 77	N 004E 78	O 004F 79
5–	P 0050 80	Q 0051 81	R 0052 82	S 0053 83	T 0054 84	U 0055 85	V 0056 86	W 0057 87	X 0058 88	Y 0059 89	Z 005A 90	[005B 91	\ 005C 92	] 005D 93	^ 005E 94	_ 005F 95
6–	` 0060 96	a 0061 97	b 0062 98	c 0063 99	d 0064 100	e 0065 101	f 0066 102	g 0067 103	h 0068 104	i 0069 105	j 006A 106	k 006B 107	l 006C 108	m 006D 109	n 006E 110	o 006F 111
7–	p 0070 112	q 0071 113	r 0072 114	s 0073 115	t 0074 116	u 0075 117	v 0076 118	w 0077 119	x 0078 120	y 0079 121	z 007A 122	{ 007B 123	| 007C 124	} 007D 125	~ 007E 126	DEL 007F 127
8–	€ 20AC 128		‚ 201A 130	ƒ 0192 131	„ 201E 132	… 2026 133	† 2020 134	‡ 2021 135	ˆ 02C6 136	‰ 2030 137	Š 0160 138	‹ 2039 139	Œ 0152 140		Ž 017D 142	
9–		‘ 2018 145	’ 2019 146	“ 201C 147	” 201D 148	• 2022 149	– 2013 150	— 2014 151	˜ 02DC 152	™ 2122 153	š 0161 154	› 203A 155	œ 0153 156		ž 017E 158	Ÿ 0178 159
A–	NBSP 00A0 160	¡ 00A1 161	¢ 00A2 162	£ 00A3 163	¤ 00A4 164	¥ 00A5 165	¦ 00A6 166	§ 00A7 167	¨ 00A8 168	© 00A9 169	ª 00AA 170	« 00AB 171	¬ 00AC 172	SHY 00AD 173	® 00AE 174	¯ 00AF 175
B–	° 00B0 176	± 00B1 177	² 00B2 178	³ 00B3 179	´ 00B4 180	µ 00B5 181	¶ 00B6 182	· 00B7 183	¸ 00B8 184	¹ 00B9 185	º 00BA 186	» 00BB 187	¼ 00BC 188	½ 00BD 189	¾ 00BE 190	¿ 00BF 191
C–	À 00C0 192	Á 00C1 193	Â 00C2 194	Ã 00C3 195	Ä 00C4 196	Å 00C5 197	Æ 00C6 198	Ç 00C7 199	È 00C8 200	É 00C9 201	Ê 00CA 202	Ë 00CB 203	Ì 00CC 204	Í 00CD 205	Î 00CE 206	Ï 00CF 207
D–	Ð 00D0 208	Ñ 00D1 209	Ò 00D2 210	Ó 00D3 211	Ô 00D4 212	Õ 00D5 213	Ö 00D6 214	× 00D7 215	Ø 00D8 216	Ù 00D9 217	Ú 00DA 218	Û 00DB 219	Ü 00DC 220	Ý 00DD 221	Þ 00DE 222	ß 00DF 223
E–	à 00E0 224	á 00E1 225	â 00E2 226	ã 00E3 227	ä 00E4 228	å 00E5 229	æ 00E6 230	ç 00E7 231	è 00E8 232	é 00E9 233	ê 00EA 234	ë 00EB 235	ì 00EC 236	í 00ED 237	î 00EE 238	ï 00EF 239
F–	ð 00F0 240	ñ 00F1 241	ò 00F2 242	ó 00F3 243	ô 00F4 244	õ 00F5 245	ö 00F6 246	÷ 00F7 247	ø 00F8 248	ù 00F9 249	ú 00FA 250	û 00FB 251	ü 00FC 252	ý 00FD 253	þ 00FE 254	ÿ 00FF 255

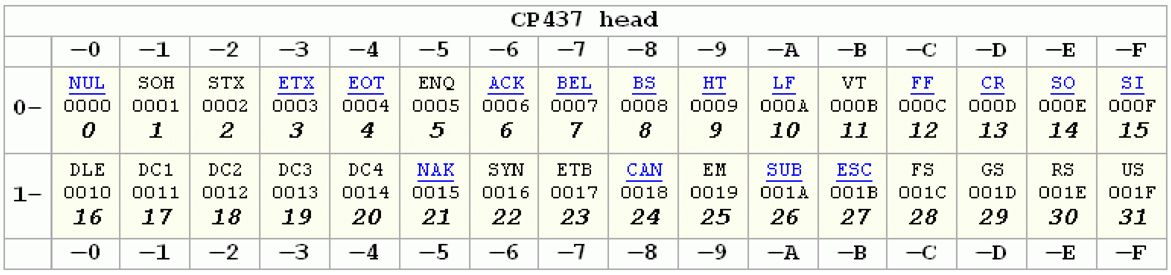
Original DOS character set

This is the original DOS character set that was used by the Saia PG3.
Code Page 437



The following is a table representing CP437 using the equivalent Unicode characters.
Standard ASCII and ISO 8859-1 (Latin-1) character glyphs, along with the Greek letters, are shown as coloured cells.
Due to the dual use of values in the range 0 to 31 (hexadecimal 00 to 20), there are two sets for these, the first being their meanings as ASCII control characters and the second their graphical output on screen/printer.
For value 127 (7F), its graphical output is shown in the last table, its meaning being the ASCII control character "DEL" (delete), Unicode value U+007F.

Legend: yellow cells are control characters, blue cells are punctuation, purple cells are numbers, green cells are ASCII letters, and tan cells are international letters.



CP437 head

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	NUL 0000 *0*	SOH 0001 *1*	STX 0002 *2*	ETX 0003 *3*	EOT 0004 *4*	ENQ 0005 *5*	ACK 0006 *6*	BEL 0007 *7*	BS 0008 *8*	HT 0009 *9*	LF 000A *10*	VT 000B *11*	FF 000C *12*	CR 000D *13*	SO 000E *14*	SI 000F *15*
1–	DLE 0010 *16*	DC1 0011 *17*	DC2 0012 *18*	DC3 0013 *19*	DC4 0014 *20*	NAK 0015 *21*	SYN 0016 *22*	ETB 0017 *23*	CAN 0018 *24*	EM 0019 *25*	SUB 001A *26*	ESC 001B *27*	FS 001C *28*	GS 001D *29*	RS 001E *30*	US 001F *31*
	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F

CP437

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	FSP 2007 0	☺ 263A 1	☻ 263B 2	♥ 2665 3	♦ 2666 4	♣ 2663 5	♠ 2660 6	• 2022 7	◘ 25D8 8	○ 25CB 9	◙ 25D9 10	♂ 2642 11	♀ 2640 12	♪ 266A 13	♫ 266B 14	☼ 263C 15
1–	► 25BA 16	◄ 25C4 17	↕ 2195 18	‼ 203C 19	¶ 00B6 20	§ 00A7 21	▬ 25AC 22	↨ 21A8 23	↑ 2191 24	↓ 2193 25	→ 2192 26	← 2190 27	∟ 221F 28	↔ 2194 29	▲ 25B2 30	▼ 25BC 31
2–	SP 0020 32	! 0021 33	" 0022 34	# 0023 35	$ 0024 36	% 0025 37	& 0026 38	' 0027 39	(0028 40	) 0029 41	* 002A 42	+ 002B 43	, 002C 44	- 002D 45	. 002E 46	/ 002F 47
3–	0 0030 48	1 0031 49	2 0032 50	3 0033 51	4 0034 52	5 0035 53	6 0036 54	7 0037 55	8 0038 56	9 0039 57	: 003A 58	; 003B 59	< 003C 60	= 003D 61	> 003E 62	? 003F 63
4–	@ 0040 64	A 0041 65	B 0042 66	C 0043 67	D 0044 68	E 0045 69	F 0046 70	G 0047 71	H 0048 72	I 0049 73	J 004A 74	K 004B 75	L 004C 76	M 004D 77	N 004E 78	O 004F 79
5–	P 0050 80	Q 0051 81	R 0052 82	S 0053 83	T 0054 84	U 0055 85	V 0056 86	W 0057 87	X 0058 88	Y 0059 89	Z 005A 90	[005B 91	\ 005C 92	] 005D 93	^ 005E 94	_ 005F 95
6–	` 0060 96	a 0061 97	b 0062 98	c 0063 99	d 0064 100	e 0065 101	f 0066 102	g 0067 103	h 0068 104	i 0069 105	j 006A 106	k 006B 107	l 006C 108	m 006D 109	n 006E 110	o 006F 111
7–	p 0070 112	q 0071 113	r 0072 114	s 0073 115	t 0074 116	u 0075 117	v 0076 118	w 0077 119	x 0078 120	y 0079 121	z 007A 122	{ 007B 123	| 007C 124	} 007D 125	~ 007E 126	⌂ 2302 127
8–	Ç 00C7 128	ü 00FC 129	é 00E9 130	â 00E2 131	ä 00E4 132	à 00E0 133	å 00E5 134	ç 00E7 135	ê 00EA 136	ë 00EB 137	è 00E8 138	ï 00EF 139	î 00EE 140	ì 00EC 141	Ä 00C4 142	Å 00C5 143
9–	É 00C9 144	æ 00E6 145	Æ 00C6 146	ô 00F4 147	ö 00F6 148	ò 00F2 149	û 00FB 150	ù 00F9 151	ÿ 00FF 152	Ö 00D6 153	Ü 00DC 154	¢ 00A2 155	£ 00A3 156	¥ 00A5 157	₧ 20A7 158	ƒ 0192 159
A–	á 00E1 160	í 00ED 161	ó 00F3 162	ú 00FA 163	ñ 00F1 164	Ñ 00D1 165	ª 00AA 166	º 00BA 167	¿ 00BF 168	⌐ 2310 169	¬ 00AC 170	½ 00BD 171	¼ 00BC 172	¡ 00A1 173	« 00AB 174	» 00BB 175
B–	░ 2591 176	▒ 2592 177	▓ 2593 178	│ 2502 179	┤ 2524 180	╡ 2561 181	╢ 2562 182	╖ 2556 183	╕ 2555 184	╣ 2563 185	║ 2551 186	╗ 2557 187	╝ 255D 188	╜ 255C 189	╛ 255B 190	┐ 2510 191
C–	└ 2514 192	┴ 2534 193	┬ 252C 194	├ 251C 195	─ 2500 196	┼ 253C 197	╞ 255E 198	╟ 255F 199	╚ 255A 200	╔ 2554 201	╩ 2569 202	╦ 2566 203	╠ 2560 204	═ 2550 205	╬ 256C 206	╧ 2567 207
D–	╨ 2568 208	╤ 2564 209	╥ 2565 210	╙ 2559 211	╘ 2558 212	╒ 2552 213	╓ 2553 214	╫ 256B 215	╪ 256A 216	┘ 2518 217	┌ 250C 218	█ 2588 219	▄ 2584 220	▌ 258C 221	▐ 2590 222	▀ 2580 223
E–	α 03B1 224	ß 00DF 225	Γ 0393 226	π 03C0 227	Σ 03A3 228	σ 03C3 229	µ 00B5 230	τ 03C4 231	Φ 03A6 232	Θ 0398 233	Ω 03A9 234	δ 03B4 235	∞ 221E 236	φ 03C6 237	ε 03B5 238	∩ 2229 239
F–	≡ 2261 240	± 00B1 241	≥ 2265 242	≤ 2264 243	⌠ 2320 244	⌡ 2321 245	÷ 00F7 246	≈ 2248 247	° 00B0 248	∙ 2219 249	· 00B7 250	√ 221A 251	ⁿ 207F 252	² 00B2 253	■ 25A0 254	NBSP 00A0 255
	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F

21.2 XOB List

Each Exception Organization Block (XOB) has a specific function.

XOB	Description	Priority
0	Power down	4
1	Power down in extension rack	1
2	Low battery	1
3	Task/Temp Data overflow	3
4	Parity error on main bus (PCD6 only)	1
5	No response from I/O module	1
6	External error	1
7	System overload	3
8	Invalid opcode	4
9	Too many active tasks (GRAFTEC)	1
10	PB / FB nesting depth overflow	1
11	COB supervision time exceeded	2
12	Index register overflow	1
13	Error flag set	3
14	Cyclic XOB	2
15	Cyclic XOB	2
16	Cold Start	2
17	S-Bus XOB Interrupt Request	2
18	S-Bus XOB Interrupt Request	2
19	S-Bus XOB Interrupt Request	2
20	Interrupt input IN0 / Interrupt input INB0 **(1)**	2
21	Interrupt input IN1	2
22	Interrupt input IN2	2
23	Interrupt input IN3	2
24		
25	Time Cyclic Alarm / Interrupt input INB1 **(1)**	2
26	Time Cyclic Alarm	2
27	Time Cyclic Alarm	2
28	Time Cyclic Alarm	2
29	Time Cyclic Alarm	2
30	RIO connection master slave	1

(1) For PCD1 and PCD2.M1xx, XOBs 20 and 21 are Interrupt inputs INB0 and INB1 respectively.

Exception Priorities

There are 4 priority levels for XOBs. Note that XOB priorities are slightly different for the older PCDs.

Level 4 exceptions (highest)
Priority level 4 is the highest priority, only XOBs 0 and 8 can interrupt execution of another XOB.

Level 2 and 3 exceptions
If a level 2 or 3 exception occurs during execution of a lower priority XOB, then it will be run directly after the end of the current level XOB.

Level 1 exceptions (lowest)
Any level 1 exception which occurs during another exception will never be handled.

Level 4 Exceptions
Priority level 4 is the highest priority, only XOB 0 and 8 can interrupt execution of another XOB.

XOB 0 Power Down
There can be up to 10ms between the call of XOB 0 and the final loss of power to the PCD to give the user time to perform some urgent saves of values.
If the XOB 0 is programmed then the message "XOB 0 START EXEC" is written into the History List at the start of the XOB and "XOB 0 EXECUTED" upon completion of the XOB, this indicates to the user that the XOB completed before power was lost.
If the XOB is not programmed then a restart cold is immediately performed upon detection of the power down. If the XOB is programmed then a restart cold is performed upon completion of the XOB if there is still power.

XOB 8 Invalid Opcode
XOB 8 is called when the firmware detects an invalid instruction in the user program.

Level 3 Exceptions
If a level 2 or 3 exception occurs during execution of a lower priority XOB, then it will be run directly after after the current level XOB.
XOB 20/25/11 have been given a higher priority level so that if the XOB is provoked during execution of a lower or equal priority then it will be executed directly after completion of the current XOB.

XOB 3 Temp/Task Data Overflow

XOB 7 System Overload
The queuing mechanism for the level 3 XOB’s has overloaded.

XOB 13 Error Flag
XOB 13 is always called when the Error flag is set by an invalid instruction, calculation, data transfer or communications error.

Level 2 Exceptions

XOB 11 COB Supervision Time exceeded
If the second line of the COB instruction indicates a monitoring time (in 1/100 seconds) and if COB processing time exceeds this defined duration, XOB 11 is called.
COB processing time is the time which can elapse between the COB and ECOB instructions.

XOB 14 Cyclic XOB
XOB 15
XOB 14 and 15 are called periodically with a frequency ranging from 5 ms to 1000s. This frequency can be set using the SYSWR instruction.

XOB 16 Cold Start
XOB 16 is the start-up XOB (Cold Start XOB), and is executed when the PCD is switched on, or is given a cold restart. XOB 16 can initialise any elements before the program begins.

If during the execution of the XOB 16 an error occurs, the XOB 13 is not called.

XOB 17 S-Bus XOB Interrupt Request
XOB 18
XOB 19
These three XOBs are started by a message on the S-Bus network; it is also possible to start them with the SYSWR instruction.

XOB 20..25 Interrupt Inputs IN0..IN3 (NT systems)
Executed on a rising edge of interrupt inputs IN0 to IN3.

XOB 20 and 25 Interrupt Inputs INB0 and INB1 (PCD1 and PCD2M1xx only)
These XOBs are called when interrupt input INB1 (resp INB2) of the PCD1/2 has detected a rising edge (see PCD1/2 hardware manual for further details).

Level 1 Exceptions

Lowest priority. Any level 1 exception which occurs during another exception will never be treated.

XOB 1 Power down in extension rack
The voltage monitor in the supply module of an extension rack (PCD 2 or PCD6) detected an excessive drop in voltage.
In this case all Outputs of the extension rack are set low within 2ms and XOB 1 is invoked.
If Outputs from this "dead" extension rack continue to be handled (set, reset or polled) by the user program in any CPU, XOB 4 and/or XOB 5 are also invoked. (Only PCD4).
XOB 1 will be called once up to 250 ms after detection of the error.
SYSWR can be used to change the behavior of XOBs 1 and 2.

XOB 2 Battery failure or low battery
The battery is low, has failed or is missing.
Information in non-volatile Flags, Registers or the user program in RAM as well as the hardware clock may be altered.
XOB 2 is called by CPU 0 every 250 ms in the event of this error.
SYSWR can be used to change the behavior of XOBs 1 and 2.

XOB 4 Parity Failure
XOB 4 can only be invoked with PCD having extension racks (PCD6 only).
The monitor circuit of the address bus has noticed a parity error. This can either arise from a faulty extension cable, a defective extension rack or from a bus extension module, or else it is simply because the extension rack addressed is not present.

XOB 5 No response from I/O module (I/O Quit Failure)
The PCD's Input and Output modules return a signal to the CPU which has addressed them. If this signal is not returned, then XOB 5 is called.
Generally, this occurs if the module is not present, but it can also happen in the case of faulty address decoding on the module.
This mechanism is not implemented on the PCD1 and 2.

XOB 6 External error
Not used. (Foreseen for intelligent modules of the PCD6)

XOB 9 Too many Graftec tasks
More than 32 Graftec branches were simultaneously activated in a Sequential Block (SB).

XOB 10 More than 7 nested PB/FB calls
PBs and FBs can be nested to a depth of 7 levels. An additional call (calling the 8th level) results in

XOB 10 executing.
The 8th level call is not executed.

XOB 12 Index Register overflow
If a program contains an indexed element which falls outside its address range (0 to 8191), then XOB 12 is called.

XOB 30 RIO connection master / slaves
After every message sent from the master to a slave, the connection is tested. If the test is not answered positively by the slave, the master CPU calls XOB 30.
This is essentially the case when, online, a station is removed from the network or closed down.