CodeWarrior Development Studio Macro-processor Language Reference Manual

Document Number: CWPEXMLREF
Rev 10.6, 02/2014

Contents

Section number **Title** **Page**

Chapter 1
Introduction

This document describes the macro-processor embedded in PE serving namely for component code generation based on the component driver. It defines the syntax and semantics of the macro-processor, as well as the symbols predefined including their importance. In addition, TST and CHG script files are described herein. This manual is composed of the following chapters:

Table 1-1. Structure of the Manual

Chapter Title	Description
Introduction	Provides the introduction of the manual.
Terms and Definitions	Describes the terms and definitions used in the manual.
Processor Expert Macro-language Description	Describes the symbols and commands for Processor Expert Macro-language.
TST Script for Component Testing	Describes the TST script files for component testing.
Component Scripts	Describes the setting control script for CHG script files
TS2 Script for Component Interdependence Testing	Describes the TS2 script file for component interdependence testing.
CDB	Describes the CDB files.
External Libraries	Describes the external libraries API.

Chapter 2
Terms and definitions

" **PE** " means an acronym for Processor Expert.

"**Source file**" means a file containing source texts for the device in question. In code generation, this file is processed using an internal macro-processor (see Processor Expert macro-language description), which will process, in a defined manner, conditional translations, replacing the symbols defined by their values. The name of this file derives from the component name, with extension that reads "src".

"**RTI**" stands for Real Time Interrupt.

"**Shadow variables**" are variables for sharing values in the write-only registers.

"**Resource**" means an internal processor peripheral (for example, pin, port, timer, serial channel, etc.), while component means a PE component (for example, "One-bit input/output" or "Date and time".)

"✕" this character means that the following symbol/convention is no more recommended for use, it is preserved only for backward compatibility and may not be supported within future PE versions.

"**template**" means a file containing blank function bodies ready for code writing by the user. PE has the capacity of generating both the interface and implementation part of this module, where the interface will contain headers and implementation will contain imports of shared items and empty function bodies.

"***CHG script file***" is a script for component setting control within OOI in the PE environment.

"***TST script file***" is a script for component setting testing prior generating, which dependant in terms of implementation.

"***CPUDB***" stands for CPU database in Processor Expert. Proprietary format.

Chapter 3
Processor Expert macro-language description

The text of the component driver is processed by Processor Expert macroprocessor. This is special macroprocessor designed for this kind of component drivers. The macroprocessor supports:

Macro is an identifier that holds any value. Identifier of a macro can contain characters: a..z, A..Z, 0..9, _ and cannot start with a digit. The value can be string or number. If macro value is a number or a string, the macro identifier can be directly replaced by its value in the driver text.

`%{def_name}, %'{def_name}', %"{def_name}", %~{def_name}~` will be replaced by its value.

Example:

```
%define MyMacro local_value

MyMacro=%MyMacro

MyMacro=%'MyMacro'_3333
```

After processing by macroprocessor the result will be:

```
MyMacro=local_value

MyMacro=local_value_3333
```

There are several types of macros:

Command starts with character % as a first non-space character on line. The command ends at the end of line.

A source file may be a driver (`*.DRV`), `TST` or `TS2` script file and `CHG` script file. Source file is processed by Processor Expert macro-processor to validate component settings or generate code.

Processor Expert defines the following symbols (macros) prior to script execution:

- Global macros or symbols - Defined by Processor Expert, the same for all components during code generation, cannot be changed by driver. Driver can define new global macro but cannot modify them. Processor Expert settings, CPU component settings, same information for all components;
- Local macros or symbols specific for Component - contains component specific settings. They are of two types:
  - Defined by Processor Expert for each component, cannot be changed by a driver. Some of them are defined for each component and the others depend on component's properties, methods, and events.
  - Defined by driver and can be changed by a driver.
- Special macros and directives or symbols for each property/method/event - information about settings for each item;

Symbol small/capital letters are distinguished; however, two identical symbols differing only by a small/capital letter in the symbol name cannot be used due to the backward compatibility. The driver-defined symbol names are subject to the conventions described herein.

Methods

Symbols determining generating methods match method names; if pre-defined, the user will require the method in the source code, otherwise, the method will not be required. Method name small/capital letters are distinguished and included in the description. The symbol values correspond to method user names or match the method symbols unless method renaming has been opted by the user.

If the target language does not support identical method names in different modules (or the object approach), a method name will be created as `%'ModuleName'_MethodName`. The "_" character positioned between the module name and method is defined using "`%.`" macro and depends on the compiler.

Events

Symbols determining handling the events in question match event names. If pre-defined, the user will require event handling in the source code, where the symbol value will correspond the name of the procedure to be executed at each event.

This chapter consists of the following topics:

3.1 Symbols

This section explains the symbols defined based on processor component and contains the following topics:

3.1.1 Global symbols

This section describes the symbols defined globally for the entire project. The topics covered here:

3.1.1.1 Processor component symbols

This topic describes the symbols defined based on processor component.

`Language`	Targets the language identification, '`ANSIC`', '`ASM`'.
`TimeStamp`	Translation time (date, time), data and time formats from the operating system (long date, short time); the symbol is intended only for information included in the comment; the comment must extend over the entire line for easier comparison of the files within PE.
`CPUvariant`	Types item within the processor property editor provided there are multiple variants in the processor; if not, the same as `CPUtype`.
✗ `CPU`	{*deprecated*} Same as the `CPUvariant`.
`CPUtype`	Processor type (component name).
`CPUfamily`	Processor family as per CPU DB.
`CPUsubFamily`	Processor sub-family as per CPU DB.

Table continues on the next page...

CPUproducer	Processor producer as per CPU DB.
CPUendian	Byte coding for CPU ('big' (for example, Freescale), 'little').
CPU_DB_version	CPU database version.
CPUmanualVersion	CPU manual version as per which CPUDB has been created.
ProcessorModule	CPU module name.
ProcessorName	CPU component name (as named by the user).
ExternalModules	List of names of external user modules that the user wishes to link to the project.
ExternalModuleExts	List of external user module extensions.
ExternalModuleDir	List of paths to external user modules; backslash is used as sub-directory delimiter and it is finished by backslash.
ExternalModuleRelDir	List of relative paths to external user modules against the code directory; ends with a slash; not supported for Eclipse - the list contains empty string instead relative path.
ExternalModuleHeader	List of Y/N values depending on whether include for the external user module exists on the disk or not.
Xtal_kHz	Processor timing frequency in kHz, integer (rounded).
Xtal_kHz_real	Processor timing frequency in kHz, real number.
[High|Low|Slow]Clock_kHz	Clock before the system prescaler in the high|low|slow speed mode, i.e. PE timing source clock frequency (timing model root), integer number; the symbols are not provided for SpeedModeList mode.
[High|Low|Slow]Clock_kHz_real	Clock, i.e. PE timing source clock frequency (timing model root), before the system prescaler in the high|low|slow speed mode, real number; the symbols are not provided for SpeedModeList mode.
[High|Low|Slow]SpeedClock	Setting for the main prescaler (item value), defined only for supported speed-modes. **NOTE:** This is a remainder of the Toshiba time module, where the timing source was pre-selected using the main prescaler; recently not in use, 1 remains; these symbols are not available for data-driven CPU components, if there are no corresponding properties in the component; the list is not provided for `SpeedModeList` mode.
CPUsystem_ticks	Number of ticks after pre-selecting by the main prescaler, i.e. main prescaler setting (a list of 3 values - for each mode); the list is not provided for `SpeedModeList` mode.
OnChipRAM	
OnChipEEPROM	
OnChipFLASH	Sizes of RAM, EEPROM and FLASH on the chip; for other memories, the size is defined only locally for the processor module.
Dir_Compiler	{*deprecated*} External compiler directory (slash-terminated absolute path), to be defined based on the tools setting; it is not defined for Eclipse.
PE_GENERATING	Defined under the following conditions: 1) generating; 2) TS2 script file processing prior generating the code; 3) TS2 script file processing, if the peripheral initialization inspector is open and all symbols for RGI processing need to be defined.

Table continues on the next page...

InstructionClock	Frequency of the instruction execution clock in MHz; a list of real numbers for all speed modes; to be defined as per the prescaler from CPU DB named InstructionClock, which, in addition, contains a special register with a function named in the same manner; if the frequency is not defined or unknown, the list will contain an undef value.
OperatingSystemId	Identification of the Operating System as per the component in the project presenting the Operating System;
O_xxx	Symbols based on the active configuration items (convention: each item symbol must start with O_).
__REG_INIT_ONLY__	RGI processing identification, this symbol is not defined in PE, it serves for crating RGI from the driver - by %ifdef __REG_INIT_ONLY__ conditional translation, or %ifndef __REG_INIT_ONLY__;

3.1.1.2 Symbols defined at all times

An error corrected in the 2.71 edition - no global symbols could be defined unless the project included CPU component. (For CPU.chg, which is not selected as a target, no symbol needs to be defined.)

PEversion	Processor Expert (core/service) version number that reads as 'XX.XX'; the version may be presented as a hex number however in the past was used only decimal digits; see also PEversionDecimal.
PEversionDecimal	Processor Expert (core/service) version number, as a decimal number; PE version is internally represented as a hex number (for example version 4.46 is internally represented as number 0x446), this symbol contains decimal representation of this number (for example, for 0x446 = 1094).
PE_ProductVersion	Processor Expert (product) version number.
PE_ProductVersionLimited	Defined only if the PE version is not a full version; it may become one of the following values: ALPHA, BETA, PROTOTYPE and DEMO; the information needs to be obtained from PE.cfg - ProductVersion.
ProjectStaticFilesGenerationMode	Performs the identification of the selected mode of creation of static files for the project (the selection is done only during project creation), supported values: LINKED (default) and STANDALONE.
Compiler	Compiler identification; the symbol as per the file name in the directory as follows: Config\Compilers*.cmplr
CommentLine	Defined if a sequence of characters exists for the language in question quoting the comment to the line ends; if this value is reached, information will be obtained from the configuration file of the compiler selected.

Table continues on the next page...

CommentBrackets	Defined if a sequence of characters exists for the language in question quoting and terminating the multiple-line comment; the sequence presents the symbol value, information will be obtained from the configuration file of the compiler selected.
DriverExtension	Durrent driver file extension; necessary for processing TST, RGI, DRV and DMO files using SRC.
ModuleList	List of names of all component modules (except for the CPU module); not defined provided no additional module exists; the symbol will be available only for generating.
EventModuleList	List of names of all potential modules; not defined if there are no modules; the symbol will be available only for generating.
✘ Dir_Project	{*deprecated*} Project directory (slash-terminated absolute path); replaced by ClientDir_Project.
ClientDir_Project	Project directory, absolute path; backslash is used as sub-directory delimiter and it is finished by backslash.
✘ Dir_Drivers, Dir_Events, Dir_Binary	{*deprecated*} Absolute target directories for generating codes for drivers, events and binary (translated) files, slash-terminated; use relative paths if possible.
✘ ClientDir_Code, Dir_Events, Dir_Binary	{*deprecated*} Absolute target directories for generating codes for drivers, events and binary (translated) files, slash-terminated; use relative paths if possible.
DirRel_Code	Relative target directory for generating code; sub-directories are separated by backslash, the path is finished by backslash.
DirRel_Binary, DirRel_Docs	Relative target directories for generating binary files and documentation, in formats specified by the user (relative in respect to the project directory, or absolute); slash-terminated if not empty; (DirRel_Binary is relative path to project even the user can enter only relative path to code directory); backslash is used as sub-directory delimiter and it is finished by backslash.
DirRel_StaticCode	Relative target directory for static code (relative in respect to the project directory, or absolute); backslash is used as sub-directory delimiter and it is finished by backslash.
✘ DirRel_EventToDrivers	{*deprecated*} Relative path from the events directory to the drivers directory, i.e. empty string, setting not possible by the current PE.
DirRel_EventToBinary	Relative path from the events directory to the binary directory, i.e. exactly what is set by the user; backslash is used as sub-directory delimiter and it is finished by backslash.
DirRel_Events	Relative path from the project directory to the events directory; backslash is used as sub-directory delimiter and it is finished by backslash.
✘ DirRel_Drivers	{*deprecated*} Relative path from the project directory to the generated code and events directory (for DirRel_Drivers is applied DirRel_Events+DirRel_EvntToDrivers); the symbol was replaced by DirRel_Code.
✘ DirRel_DriverToEvents	{*deprecated*} Relative path from the drivers folder to the events folder (empty string: current PE does not allow the setup).
DirRel_BinaryToEvents	Relative path from the binary directory to the events directory (in most cases, "" or "..\"); backslash is used as sub-directory delimiter and it is finished by backslash.

Table continues on the next page...

✘ `DirRel_Sources`	{*deprecated*} Relative path of main & events directory selected by user, this symbol is available only in Eclipse with Java PE service; should be replaced by `DirRel_Code`.
`DirRel_ProjectSettings`	Relative path of Project settings directory selected by user, this symbol is available only in Eclipse with Java PE service; backslash is used as sub-directory delimiter and it is finished by backslash.
`Dir_PE`	PE system directory (slash-terminated absolute path).
`ServerDir_PE, ClientDir_PE`	Root PE directory on server and on client (absolute path, finished by backslash); for stand-alone PE, PE Java service and CW plug-in both symbols are the same; backslash is used as sub-directory delimiter and it is finished by backslash.
`PE_DemoVersion`	Indicates PE demo version.
`PE_DEVELOPMENT`	Eclipse: PE development mode is tuned ON, for example, "CDE mode".
`PE_ECLIPSE`	Symbol is defined only in PE plug-in for Eclipse (correctly saying symbol is defined in PE server version for Eclipse or Java PE-service).
`ProjectName`	Project name; for Eclipse the value is ProcessorExpert for projects created in older version (to preserve backward compatibility); otherwise it is name of the Eclipse project (might not be an identifier, only character % is replaced by underscore character).
`EclipseProjectName`	Name of the Eclipse project; the name may contain relative path in the workspace; the symbol may not be available during project creation or project closing; recommended to use symbol `ProjectName` instead.
`ProjectModule`	Main project module name generated from the "Main.src" file; this module usually contains only import of processor module and empty main function (a user template); deprecated, replaced by `ProjectMainModule`.
✘ `ProjectMainModule`	{*deprecated*}Main project module name generated from the `Main.src` file; this module usually contains only `main()` function.
`SetHighSpeedMode`	Within CPU, HighSpeedMode is supported (always defined) - at the same time, this is a mode change method name, the value stands for the mode name (main clock source) as it is selected in the corresponding property, empty string If there is no property for clock source selection; the symbol is not available for SpeedModeList mode, list `SpeedModeList` should be used instead.
`SetLowSpeedMode`	Within CPU, `LowSpeedMode` is supported - at the same time, this is a mode change method name, the value stands for the mode name (main clock source) as it is selected in the corresponding property, empty string If there is no property for clock source selection; the symbol is not available for `SpeedModeList` mode, list `SpeedModeList` should be used instead.
`SetSlowSpeedMode`	Within CPU, `SlowSpeedMode` is supported - at the same time, this is a mode change method name, the value stands for the mode name (main clock source) as it is selected in the corresponding property, empty string If there is no property for

Table continues on the next page...

	clock source selection; the symbol is not available for `SpeedModeList` mode, list `SpeedModeList` should be used instead.
`CPUrunSpeedModeNum`	A number of speed modes supported within CPU.
`SpeedModeList`	List of all supported speed modes [`SpeedMode0,SpeedMode1,...,SpeedMode7`], number of items in the list matches with `CPUrunSpeedModeNum`; the list is available only in `SpeedModeList` mode.
`SpeedModeNames`	All speed mode names: `High`, `Low`, `Slow` or `SpeedMode0`, `SpeedMode1`, ..., `SpeedMode<N>`, where <N> is index of last selected speed mode.

3.1.2 Component module symbols

This topic describes the symbols defined for each component module.

`DeviceType`	Component type, corresponding to the component name within PE.
`DeviceTypeCount()`	A number of components of the given type included and enabled in the project (current project configuration); the function is available for component scripts only.
`DeviceName`	User-defined component name unique within the project (in future, this may be more than just an identifier).
`ModuleName`	Module name, so far identical with `DeviceName`, 100% identifier.
`Comment`	Comments to the component, list of strings, component header generating symbol, not defined unless a comment is written by the user.
`runHighSpeed`	If the device is capable of running in `HighSpeedMode`; the symbol is not available for `SpeedModeList` mode, `runSpeedMode` should be used instead.
`runLowSpeed`	If the device is capable of running in `LowSpeedMode`; the symbol is not available for `SpeedModeList` mode, `runSpeedMode` should be used instead.
`runSlowSpeed`	If the device is capable of running in `SlowSpeedMode`; the symbol is not available for `SpeedModeList` mode, `runSpeedMode` should be used instead.
`runSpeedMode`	List of supported modes (Yes/No identifiers).
`runSpeedModeNum`	Number of supported modes.
`<method>`	Names of all methods requested by the user.
`<method>_Hint`	Hint for the method in question - header declaration.
`<method>_HintHint`	Hint for the method in question - description according to the component (reduced).
`<method>_HintLong`	Hint for the method in question - description according to the component (unreduced).

Table continues on the next page...

<event>	Names of all events requested by the user; the value stands for the user implementation (procedure name).
<event>Prior	Event priority (>= device priority), only defined if the event has a priority.
<event>Module	Module of event implementing (replaces the EventModule).
<event>_Hint	Hint for the event in question - header declaration.
MethodList	List of names of all methods requested.
MethodHints	list of hints (headers) of all methods requested.
EventList	list of names of all events requested.
EventModules	List of all component event modules (replaces EventModule); the list is empty if event modules are not generated; the symbol will be available only during generating. In shared modules, this will contain event modules of all components within the project.
InhrSymbolList	List of symbols of all inheritance items with components generating code (a component module) and all linked components.
ComponentVersion	Component version.
ComponentTemplate	Component template, defined for templates only.
ComponentUserCopyright	User copyright for the component (only defined if specified in .bean), a list of lines to be generated within the header.
ComponentAllocatedLicenses	The list is available if the component options contains feature "AllocateAllLicenses" and contains all licenses, that were allocated; as the licenses are allocated only for code generation, the symbol is also available only during code generation - it should not be accessed from TST or CHG scripts, use PE_GENERATING symbol to check if it is code generation in progress.

3.1.3 CPUDB functions

This section describes the CPUDB functions and covers the following topics:

- Registers
- CPU package and pins
- Memory Map
- Others

3.1.3.1 Registers

`CPUDB_get_register_bit_name(reg,num)`	Stands for a register bit name, reg stands for a register name, num stands for a number of bit from zero to the highest bit; if the num refers to a bits group, function returns `<BG><i>` where `<BG>` is name of the bits group and `<i>` is index of a bit in the bits group (starting from zero); an error return an empty string (**NOTE:** Any unused pins are stored under "Unused" within CPUDB).
`CPUDB_get_register_bits_name(reg,num)`	Stands for register bit names, reg stands for a register name, num stands for a number of bit from zero to the highest bit; an error will return an empty string (**NOTE:** Any unused pins are stored under "Unused" within CPUDB)
`CPUDB_get_register_bits_size(reg,num)`	Stands for register bit width, reg stands for a register name, num stands for a number of bit from zero to the highest bit; an error will return an empty string.
`CPUDB_get_register_bit_offset(reg,bitname)`	Bit position/positions within the register (from zero), reg stands for a register name, bitname stands for a bit or bit group name; an error will return an empty string.
`CPUDB_get_register_bit_mask(reg,bitname)`	Returns a value of `(1 << %CPUDB_get_register_bit_offset(reg,bitname))` expression; an error will return an empty string; should not be applied to bit groups.
`CPUDB_get_register_bits_mask(reg,bitsgroupname)`	Returns a mask for a bit group (or even a single bit); an error will return an empty string.
`CPUDB_get_register_bit_reset(reg,num)`	Bit value/values following resetting within the register (from zero) as per `CPU-DB`, except for 0 and 1 values, this may return the following codes: P (0 after supply, other reset will not change the value), Q (1 after supply, not changed by other reset), C (mode-dependant), ? (not defined), reg stands for a register name, num stands for a number from zero to the highest bit; if num ,all' the function will return a value (binary number) for every bit within the register (the lowest bit on the right); an error will return an empty string.
`CPUDB_get_register_bit_access(reg,num)`	Access to a bit or bits within the register (from zero), reg stands for a register name, num stands for a number from zero to the highest bit; if num ,all' the function will return a value for every bit within the register (the lowest bit on the right); possible values: R (read-only), W (write-only), X (read-write), U (unused), M (mode dependant), F (register in flash, read-only), E (register in flash, no access), 1 (read/write once), V (write-once); an error will return an empty string.
`CPUDB_get_register_bit_hint(reg,num)`	Stands for a register bit description, reg stands for a register name, num stands for a number of bit from zero to the highest bit; an error will return an empty string.
`CPUDB_get_register_bits_hint(reg,num)`	Stands for register bit descriptions, reg stands for a register name, num stands for a number of bit from zero to the highest bit; an error will return an empty string.
`CPUDB_get_register_width(reg)`	Returns register width - a decimal number; an error will return an empty string (if the register does not exist).
`CPUDB_get_register_addr(reg)`	Returns register address - a decimal number; an error will return an empty string (if the register does not exist).
`CPUDB_get_register_addr_offset(reg)`	Returns register address without register base; if register not found, returns empty string.

Table continues on the next page...

CPUDB_get_register_unused_bits_mask(reg)	Returns mask (as a decimal number) of all reserved bit-fields in register passed as command parameter; reg stands for a register name; on error, will return an empty string (if the register does not exist)
CPUDB_is_register_in_FLASH(reg)	Returns yes/no if the register is in FLASH; an error will return an empty string (if the register does not exist).
CPUDB_is_write_once_register(reg)	Returns yes/no if the register contains at least a single write-once bit; an error will return an empty string (if the register does not exist).

3.1.3.2 CPU package and pins

CPUDB_get_total_number_of_pins()	Returns a total number of pins on the CPU packaging.
CPUDB_get_number_of_pins_horiz	Returns a total number of pins on a single horizontal edge of the CPU packaging.
CPUDB_get_number_of_pins_vert	Returns a total number of pins on a single vertical edge of the CPU packaging.
CPUDB_is_pin_clock_numbering()	Returns Y/N for a clockwise packaging pin numbering sequence.
CPUDB_get_first_pin_side()	Returns U/R/D/L location (side of the packaging) of the first pin.
CPUDB_get_first_pin_pos()	Returns EGDE/CENTER position of the first pin on the packaging edge.
CPUDB_get_pin_name_by_number(num)	Returns a pin name by a number on the CPU packaging.
CPUDB_get_pin_channel_number(<cht>,<pin>,<prph>)	Returns a number of the channel for the pin entered with respect to the peripheral entered (decimal number), <cht> will specify a type of channel, it may achieve the following values: I/O (for example, pin of port) or A/D (for example, input channel of ADC) or IRQ (for example, pin of external interrupt of "IRQ" type); the pin will specify an active name of existing pin (if one exists - internal error); if the name does not exist, the peripheral will return -1; <prph> stands for peripheral name (it is preferred to use active user name, but can be used also default CPUDB name).

3.1.3.3 Memory Map

CPUDB_get_[unused_]memory_block_count()	Returns a number of memory blocks within the memory map as displayed in the memory map window (full version inc. I/O and external space). The unused_ function works with non-allocated memory only.

Table continues on the next page...

`CPUDB_get_[unused_]memory_block_info(index, info-type)`	Returns the requested information on the memory block with index specified by parameter index (1 up to the number of blocks), info-type is `ADDRESS` (initial block address), `SIZE` (block size in addressable units), `TYPE` (memory type: `RAM`, `ROM` (i.e. `ROM` or `OTP`), `FLASH`, `I/O`, `EEPROM`, `FIRMWARE`, `EXTERNAL` i.e. unoccupied external address space), `DISPLAY_TYPE` (memory type as displayed in the User Interface, standard memory type may be overwritten by custom value), `AREA` (name of memory area if there are multiple supported by `CPU`; if not supported, this will return an empty string; naming will depend on the `CPU` model, largely `CODE` or `DATA`), `MIRROR` (returns yes/no, depending on whether this is a memory block that mirrors a memory of another block), `MIRROR_FROM_ADDR` (returns the address of source address, "n/a" if the block is not mirrored), `WORDSIZE` (the width of addressable block word in bits), `EXTERNAL` (will return yes/no, depending on whether this is a block outside the internal memory); `SDRAM` (returns yes/no, depending on whether this is `SDRAM`-like `RAM`); `StandByRAM` (will return yes/no, depending on whether `RAM` is battery backed-up `RAM`); `NAME` (returns specific name of the memory block, usually it is unused; used for "`BootROM`"); `HINT` (will return the hint to the memory block that has been assigned to that block under the PE code, contains additional information regarding the block and is displayed in the hint within PE); the `unused_` function will work with non-allocated memory only.
`CPUDB_get_[unused_]memory_block_index(addr[,space])`	Returns the memory block index within the memory map according to the address specified (from one); in case of error, it will return zero; the optional space parameter specifies the name of the area in which the address is to be searched (implicitly, the main address area), area names are displayed within the memory map heading; the `unused_` function will work with a non-allocated memory only.

3.1.3.4 Others

`CPUDB_get_allocated_bean_type(prph)`	Returns a component name (DeviceType) that allocates the device specified, prph is a default device name as per CPU DB; if the peripheral does not exist, PE will report an internal error; in the case of other error (no CPU component selected, peripheral not allocated) the function will return an empty string.
`CPUDB_get_allocated_bean_name(prph)`	Returns a user component name that allocates the device specified, prph is a default device name as per CPU DB; if the peripheral does not exist, PE will report an internal error; in the case of other error (no CPU component selected, peripheral not allocated) the function will return an empty string.

3.1.4 Symbols for driver

This section describes the symbols defined by each driver, as local symbols, i.e. by %define command.

`DriverVersion`	Driver version number that reads as XX.XX
`DriverAuthor`	Driver author name
`DriverDate`	Driver modification last date that reads as DD.MM.YYYY

3.1.5 Component functions

`setVariable(name,value)`	Assigns the variable (with the specified value) to the current component. The variable exists with the component instance; it is not affected if the component is disabled, but for future extension it is not guaranteed that the variables will be preserved for disabled components. All variables are removed if variant of the peripheral initialization component is changed. It is not stored into the project. The function returns empty string. The variable name must be an identifier. The variable names starting with `PE_` and `PEx_` prefixes are reserved for interaction with Processor Expert core.
`getVariable([@comp@]name[,default-value])`	Returns value of the component variable; if the variable does not exists, returns default-value; if default-value is not specified generates script error; optional `@comp@` prefix can be used to refer variables from other components (via component name), prefix `@_CPU_@` can be used to refer processor component.
`delVariable(name)`	Removes the component variable; if variable does not exists, returns string `not_found`, otherwise returns empty string.

Listing: Example - Count Execution of Script

```
%setVariable(exec_counter,%EXPR(1+%getVariable(exec_counter,0)))
%hint COUNTER = %getVariable(exec_counter)
%if (%getVariable(exec_counter) == "1000")
  %delVariable(exec_counter)
%endif
```

Reserved script variables for Processor Expert Core:

PEx_RequiredInitMethodName	Defines with non-empty value, if the component initialization is not called after reset, value contains name of the method, that needs to be invoked for the initialization.
PEx_RequiredPinRoutingInitMethodName	Defines with non-empty value, if the pin-routing initialization is not called after reset; value contains name of the method, that needs to be invoked for the initialization; this variable is used to identify "dynamic" configuration for pins model.
PEx_RequiredPeripheralsModelInitMethodName	Defines with non-empty value, if the peripheral initialization is not called after reset; value contains name of the method, that needs to be invoked for the initialization; this variable (if defined) is used to identify "dynamic" configuration for property model.
PEx_InitSequenceComment	Contains comment displayed for the component in the InitializationSequence view; optional value.
PEx_ClkSigOutFreqTxt_<element>	Reserved for interaction with clock diagrams custom view plug-in; specifies text to be displayed as output frequency for clock signal element in clock flow diagram; <element> is a name (identifier) of the clock signal element.
PEx_GeneratesComponentModule	Used to disable generation of the component module using %INTERFACE and %IMPLEMENTATION; acceptable value is "no"; this information may be also specified statically in the Options section of .bean file.
PEx_PinAndPropertyModelDynamicConflictResolutionMsg	Contains extension of dynamic (run-time) error message displayed for the component; the component may define the text to specify recommended resolution for the conflict.
PEx_ImplementationModules	comma separated list of the component implementation modules including project relative path and file extension; “/” slashes are used to separate directories in the path (backslashes shall not be used); it is used if the implementation module does not match with component name, usually if static implementation module is provided from the component; the variable is used in Processor Expert to display code of the selected method; for example, GeneratedCode/UART3.c, GeneratedCode/UART3.h
PEx_ImplementationModules4Method{mthd}	Same as PEx_ImplementationModules, but content is applicable only for method identified by symbol {mthd}.
PEx_ComponentMethodFirstParam	Value of first method parameter (valid for all component methods); used for drag'n'drop method invocation in user code.
PEx_PddMethodFirstParam	Value of first PDD/PESL parameter (valid for all PDD/PESL component commands); used for drag'n'drop command invocation in user code.
PEx_ NoModulePrefix	the variable is alternative way to component option NoModulePrefix, how the component may specify, that generated code does not contain module prefix in the method name. If the variable is defined, the methods are generated without prefix. Value of the variable is ignored. This option does not affect code generation, the component script must ensure the corresponding code is generated.

3.2 Source file syntax for macro-processor

The source file is processed using the internal macro-processor which will produce a source file (driver), or even more files in the language required (C, assembler, Java).

This section includes the following topics:

3.2.1 Denotation

`<def_name>`	Identifier of the symbol defined (see `%define`), the symbol name is case-sensitive.
`<string>`	String starting with a symbol of quotes or a single quote or also back quote or tilde and ending with the same character; if it starts with tilde (~), macro substitution is applied to its value.
`<def_value>`	Text up to the end of line (the first space will be ignored), presents a value of the symbol defined (see `%define`).
`<def_list>`	Special list-like symbol, may contain string items.
`<number>`	Decimal number

`<item-ref>`	Reference to inspector item in one of the following formats:
	`item-symbol`
	`@_CPU_@item-symbol`
	`@component-name@item-symbol`

`<binary_operator>`	See also Math Operations **+** Summation **-** Subtraction ***** Multiplication **/** Real division **** Integer division **:** Modulo, remainder of integer division, binary operator only, this binary operator was replaced by the : symbol to avoid ambiguity the original symbol can be used as an unary operator **|** Bit or, logical count, integer operation **&** bit and, logical multiplication, integer operation **~** bit xor, integer operation **^** power (only a non-negative integer may be an exponent)

Table continues on the next page...

	> or >> bit shift to the right, integer operation
	< or << bit shift to the left, integer operation
	Binary operator priority (top to bottom):
	^
	*** / \ :**
	+ -
	> >> < <<
	&
	~
	|
`<unary_operator>`	**=** Assignment, works in the same way as %define!
	<binary_operator>= See binary operator description
	$= Converts floating point number to closest integer by rounding
	.= Converts floating point number to integer by removing of the decimal section (truncate)
	@= Logarithm (inversion operator to ^), the parameter on the left is the logarithm of the one on the right is the base, (8@=2 will return 3), the result is a real number
`<expression> <number>`	Macros and built-in functions in the format %func(params)will be accepted as well, for example, %get; the mark can be changed by superposing the minus mark, for example, `-%tmp;` `(<expression>)<expression><binary_operator><expression>`
`<pvalue>`	`<def_name>` or `<string>` or `%"def_name"` or list element (see `macro %[`)
`<condition>`	1. `defined(<def_name>)` - Testing if the symbol has been defined; it is also possible to test inherited symbols using: defined(`@<InhrItem>@<def_name>`), where `<InhrItem>` is a symbol of property for inheritance, but it is not recommended practice to create dependency on symbols from another component. 2. `ndefined(<def_name>)` - Testing if the symbol has not been defined. 3. `<pvalue> = <pvalue>` 4. `<pvalue> == <pvalue>` 5. `<pvalue> <> <pvalue>` 6. `<pvalue> != <pvalue>` 7. `<pvalue> > <pvalue>` 8. `<pvalue> < <pvalue>` 9. `<pvalue> >= <pvalue>` 10. `<pvalue> <= <pvalue>` 11. `<pvalue> =N <pvalue>` 12. `<pvalue> ==N <pvalue>` 13. `<pvalue> <>N <pvalue>` 14. `<pvalue> !=N <pvalue>` 15. `<pvalue> >N <pvalue>` 16. `<pvalue> <N <pvalue>` 17. `<pvalue> >=N <pvalue>`

Table continues on the next page...

18. `<pvalue> <=N <pvalue>` - *N* is figure 0..9 or full stop, strings will be justified before comparison (i.e. cut or spaces will be added to the left) to the length of N characters; if N is zero, the shorter string will be expanded to match the length of the longer string; if N is full stop, both strings will be converted to a real number.
19. `<pvalue> =^ <pvalue>` - Prior comparison, both strings will be converted to capital letters.
20. `<pvalue> =$ <pvalue>`
21. `<pvalue> ==$ <pvalue>`
22. `<pvalue> !=$ <pvalue>`
23. `<pvalue> <>$ <pvalue>` - Performs the comparison of strings; comparison with the operator without the symbol $; the only difference at the end: in the internal version, the operator without $ will report an error that integer and real number is being compared, while the operator including $ will not contain such control (the control serves as warning on frequent errors).
24. `<condition> | <condition>` - Logical count (or), <condition> may not contain logical multiplication, the operator cannot be combined with logical multiplication, processed using shortevaluation; This expression is not recommended, brackets should be used.
25. `<condition> & <condition>` - logical multiplication (along with), <condition> may not contain logic count, the operator cannot be combined with logical count, processed using short evaluation; This expression is not recommended, brackets should be used
26. `for_last` - Testing if the current cycle variable contains the last field item.
27. `for_lastmod4` - Testing if the current cycle variable contains the last field item or its index can be divided by 4.
28. `for_lastmod8` - Testing if the current cycle variable contains the last field item or its index can be divided by 8.
29. `for_lastmod16` - Testing if the current cycle variable contains the last field item or its index can be divided by 16.
30. `<string>` in `<def_list>` - Testing if the string is contained within the list, `<def_list>` must be a variable.

`<boolean expression>`

`<condition>`

`(<boolean expression>)`

`(<boolean expression>) & (<boolean expression>)`

`(<boolean expression>) | (<boolean expression>)`

The priority of & and | operators is not defined and needs to be treated using brackets. The Boolean expression can be divided even to multiple lines; however, the end of line must be behind the & or |operators.

To be evaluated only partially using a so-called short evaluation.

The following topic is covered here:

- Math Operations

3.2.1.1 Math operations

All symbols in macro-language are stored as a string; however there are supported several math operations as well. Macro-language allows to work both with real-numbers (with double precision) and integer-numbers (unlimited width from version PE 5.02).

For integer operations, string is converted to integer number. If the string represents real number with non-zero fractional part, error is reported.

For operations, that are supported with both real and integer operands, the real numbers are preferred. If both operands are integers, operation is executed above integer type.

3.2.2 Macro commands

A macro command must start at the beginning of the line (% as the first non-space character on the line); this line may contain only this command. The remainder of the line is a comment. For a macro command, a comment up to the end of line can also be used in the following format: `%--`.

A list of macro commands by type follows under these topics:

- Conditional translation
- Symbol definition
- Inserting files, calling external DLLs
- Text formatting and comments
- Output language and translator
- Errors
- Working with lists
- Sections
- Code identification
- Working with component items
- Expressions
- Debugging
- Insertion marks
- %set ReqValue <value>=<reason>

3.2.2.1 Conditional translation

The following listing displays the conditional translation:

Listing: Conditional translation

```
%ifdef <def_name>    conditional translation
%ifndef <def_name>   (<def_name> can be macro as well)
%if <condition> [%-comment]
%if (<boolean expression>) [%-comment]
%else
%elif <condition> [%-comment] as a %else/%if sequence
%elif (<boolean expression>) [%-comment] as a %else/%if sequence
%endif
```

3.2.2.2 Symbol definition

`%define <def_name> <def_value>`	Defines a symbol for the given driver, i.e. scope of applicability will be limited to the framework of the given file, the symbol possess `<def_value>value`; from `<def_value>`, all spaces at the start and at the end of this value will be removed with subsequent replacement of macros.
`%define! <def_name> <def_value>`	The same as with `%define`, but if a symbol of the same name exists (and is a local symbol, i.e. driver-defined), the symbol will be re-defined instead of invoking an error; from `<def_value>`, all spaces at the start and at the end of this value will be removed with subsequent replacement of macros.
`%define_prj <def_name> <def_value>`	Defines a symbol for the project as such, the symbol value will be `<def_value>`; will not work for `TST` and `CHG` scripts, when only a local symbol will be defined (while the local symbols from main.tst will be available for starting `TST` component as well); the symbol name can be defined using macros, for example, `iv%IntDevice`; from `<def_value>` all spaces at the start and at the end of this value will be removed with subsequent replacement of macros; `define_prj`. Allows for re-defining the formerly defined value, which should be, however, used only exceptionally.
`%undef <def_name>`	Deletes the definition of a driver-defined local symbol/list, if the symbol does not exist, error will be reported.
`%undef! <def_name>`	If a local driver-defined `<def_name>` symbol/list exists, this will delete the definition of such symbol/list.
`%compute_fraction(<def_name>, <real_value>, <number>)`	Calculates a fraction, which is the closest, by its value, to the real number specified and define the output into the `<def_name>_mul` and `<def_name>_div` symbols (if the symbols exist, they will be re-defined) - when both such

	values must be lesser or equal to integer <number>; the function operates with 32-bit signed integer only; serves for generating a timer driver.

3.2.2.3 Inserting files, calling external DLLs

`%include <filename>`	Insert a `<filename>` file into the text; the file name is the text up to the end of the line. The file name is entered including the directory relatively with respect to the Drivers\directory. A file name cannot contain a for cycle variable.
`%include <filename> (<par1>,<par2>,..)`	The same as with `%include`, in addition, defines the parameters transferred as those available via `%1, %2, ..macros <par?>` All characters between "`(`" and/or "`,`" and "`,`" and/or "`)`"; the parameters may not contain quote or bracket characters.
`%include><indent>`	As above, plus inserts a number of spaces specified before each line when pasted (similar as `%inclSUB>X`).
`%inclSUB[><indent>] <subroutine> [(<par1>,<par2>,..)]`	Inserts a code from a sub-program with `<subroutine>` identification to the current location of the file source, defines forwarded parameters as those available via %1, %2, .. macros <par?> or via identifiers of parameters defined under `%SUBROUTINE`, where the value of such parameters includes all characters between "`(`" and/or "`,`" and "`,`" and/or "`)`"; `<indent>` is an optional indent parameter - a decimal number providing a number of spaces to be inserted at the start of every sub-program line; if not provided otherwise, the spaces are inserted according to the indentation of the %inclSUB command; the current indent is specified by the `%SUBROUTINE_INDENT` macro.

`Comments for forwarding parameters to the sub-routine:`	If the number of parameters forwarded prior macro replacement is equal to that of the sub-routine parameters, then the parameters are separated first, and macro replacement is applied subsequently. This will allow for forwarding parameters containing characters like quotes or enclosing brackets, for example,
	`%SUBROUTINE showhint(hintmsg)`
	`%hint %hintmsg`
	`%SUBROUTINE_END`
	`%define tmp Long hint including comma, and round brackets ) ()`
	`%inclSUB showhint(%tmp)`

`%inclSUB>>>`	A special variant of the `%inclSUB` command serving for use of the sub-routine without forced indent into the code with a forced indent (see `%>>> and %<<<commands`): inserts the sub-routine with the indent defined by these commands without applying the forced indent within the sub-routine

Table continues on the next page...

	inserted this way; other command parameters including execution are identical as these of the preceding command variant.
%launchDLL	<DLLname>,<functionname>[,<parameters>] calling external DLL; <actionname>- this parameter is not supported for MCU10; <DLLname> may also contain a relative path to the PE or Windows system directory (for example, Drivers\SW\DLL\MyDllDriver.dll or Components\MyComponent\DLL\MyDllWizard.dll), specifying the .dll extension is optional; <functionname> is the name of the function to be called, function prototype: function <functionname>(Params:pchar; var MacroCmds, NewDefines:pchar):pchar; register far; or function <functionname>_STD (Params:pchar; var MacroCmds, NewDefines:pchar):pchar; stdcall far;*Optionally, arbitrary parameters can be forwarded to the function, see* <parameters>*. The parameters are forwarded to the function in the form specified within the line via the* %launchDLL*command in the* Params *parameter. The function will return the output that can be included into current active section of the driver, lines to be separated by CR/LF (this output will not be processed by the macro-processor); including, for example,* NULL*. Optionally, the function will return in the* MacroCommands *parameter commands for the macro-processor (lines separated by <CRLF>) to be executed immediately upon finishing DLL (may include the* %define or %set*commands); however, nothing will be generated into the output. Not implemented so far:Optionally,* NewDefines*contains definition of new symbols in the form as follows: NAME=VALUE<CRLF>; old values will be re-defined.* *If DLL contains the* SetParent *function, the function is called with the Handle parameter of the main application window:*procedure SetParent(Handle:HWND); register far; procedure SetParent_STD(Handle:HWND); stdcall far;
%launchExt	<ExternalObjectRelFileName>,<function-name>[,<parameters>] invoke function from external object (either DLL, shared object, JAR class or class); this command is supported only on Eclipse pure Java PE-service; <ExternalObjectRelFileName> is relative path and file name of the Windows DLL library, Linux shared object or JAR class, file extension is optional, path is relative to component directory, Processor Expert root directory or system directory and <function-name> is name of function, that will be invoked and <parameters> is optional list of parameters passed to the function. If the external object extension is not specified, firstly JAR library is looked for, secondly running OS native dynamic library extension is used (.dll on MS Windows or .so in Linux). Also, <function-name> may not end with suffix _STD as this suffix is used internaly to distinguish calling convention of external object function. For more details see External Libraries.
%checkList	(<list_name>,<new_list_name>,<title>,<header>[,<hint_list>][,AllowOnlyOne]) displays a check box dialogue for each item of the list_name list with the title sub-title and the header sub-title, if the new_list_name list had

Table continues on the next page...

	been defined prior starting the dialogue, it must contain the list values that should be checked by default (if the list is not defined, all values will be checked implicitly); once the dialogue is terminated, the `new_list_name` list will be defined/re-defined for all items that are left checked by the user; the optional `hint_list` list can contain a description of bubble tips for each item (the number of v items must match that in the `list_name` list). Due to the optional `AllowOnlyOne` parameter, only a single value can be selected out of the offer.
`%msg`	Command description is below.

3.2.2.4 Text formatting and comments

`%+[<string1>[<string2>]]<text>`	Adds `<string2><text>` at the end of the previous line provided the resulting line does not exceed DD characters, where DD is a constant defined according to the compiler. Otherwise, a separate line will be created containing `<string1><text>`. The string specification is optional; the strings are written enclosed in single quotes, however, they are generated into the text without the quotes.
`%++<text>`	Adds `<text>` to the end of the previous line regardless of the length of the resulting line.
`%+#[<num>]<string1>[<string2>]<text>`	Operates in the same way as %+, in addition, this splits the text automatically into the block by words (word separators: space, full stop, semicolon, exclamation mark, question mark, colon, comma, right bracket); the string1 parameter is mandatory. The optional num parameter provides the length of the resulting line, on which the text is to be broken.
`%substring <def_name>,<cislo1>,<cislo2>`	From the `<def_name>` string will create a sub-string starting with `<cislo1>` and length `<cislo2>`. The first character has an index `1`.
`%>>>`	Forced indent of the generated code: increases the required level of nesting of the generated code; implicitly, the level is not set and no alignment will be carried out; the first use of the command will set the level to column 0, with increments by 2 with every additional use; setting the level will cause PE enforce the alignment of the generated code to the set column (addition or deletion of excess spaces); the script will have to ensure returning the level back at the end of the generation process (even at the end of each method); the alignment will be carried out only for 'common' code generating, insertion marks will be not involved.
`%<<<`	Opposite function compared to the `%>>>` command.
`%><number>`	(Not a command; can exist anywhere within the line) Carries out the alignment of the generated text within the column specified (the first column is numbered 1), alignment to the right as well as to the left will be functioning (i.e. the directive has the capacity of inserting/deleting excess spaces), checks

Table continues on the next page...

	if at least a single space has been inserted; the number can be provided enclosed in quotes or single quotes for separating from the following text.
`%>>`	(Not a command; can exist anywhere within the line) Carries out alignment of the comment to the user-defined position (default 40), in the case of the 'do not generate the comment' option, the remainder of the line will be ignored.
`%-`	A comment up to the end of line, can only be used as a command, i.e. the "%" character must be the first non space character within the line
`%replaceGenLineEnd <string1>, <string2>[, <string3>]`	Substitutes, at the end of the last generated (i.e. previous) line, string1 for string2; if string1 does not exist, a new following line string3 (with an indent matching that of this line)will be added; if string3 is not entered, an error will be reported; **NOTE:** this command has been designed for generating well-arranged structures by the driver generator; substitution of the macros for macro values is not carried out within the strings.
`%_space_`	Macro to be replaced by a space (not a command).

3.2.2.5 Output language and compiler

`%;`	Generates the symbol as per the compiler in question quoting the comment up to the end of line within the source text of the language, the same as with the `%CommentLine` global symbol.
`%>>`	(Not a command; can exist anywhere within the line) A separator of a module name from the name of the method defined as per compiler ; it is an underscore character in most languages.
`%{`	(Not a command; can exist anywhere within the line) will generate a symbol as per the compiler in question quoting the beginning of a multiple-line comment in the source text of the language.
`%}`	(Not a command; can exist anywhere within the line) will generate a symbol as per the compiler in question quoting the end of a multiple-line comment in the source text of the language.

3.2.2.6 Errors

%error <error_msg>	Generates an error message to be displayed in the error window, error text will read as <error_msg> until the end of line.
%error!<error_msg>	Same as with the %error, with the difference of displaying an error message inc. file name and line #, terminating the generating for the given driver, designed for indication of internal errors, for example, in the driver file (mostly an unfulfilled driver expectation which should not have taken place); the error text will be displayed only in the internal PE version, the full version will display only a general error message.
%error^<error_msg>	As with %error, in addition, this will terminate the generating for the given script (for example, driver).
%warning <error_msg>	Generates a warning to be displayed in the error window, the warning text will read as <error_line> until the end of the line; if a TST script file exists for the driver, the driver will not generate messages (the developer version generates messages denoted as "{{{HIDEN}}}"); the %warning! Command will generate the message at all times.
%hint <error_msg>	Generates a hint to be displayed within the error window, the hint text will read as <error_line> until the end of the line. If a TST script file exists for the driver, the driver will not generate messages (the developer version generates messages denoted as "{{{HIDEN}}}"); the %hint! Command will generate messages at all times.
%log <message>	Adds a message into Console view; the command is designed for debugging purposes. %log Starting code generation, %define result successfully, %result Code generation finished
%exit^	Immediately terminates any additional processing (generating) of the script.

3.2.2.7 Working with lists

%add <def_list> <def_value>	Adds the <def_value> item into the <def_list> global list and create a list if none has been defined, the item must not be an empty string, automatic checking for duplicates: if the item already exists on the list, it is not added for the second time, the (<def_list>) list is a symbol valid for the project as such, however, no list created by PE can be modified; prior adding the <def_value>, all spaces in the beginning / at the end of this value will be removed and macros substituted subsequently; for global symbol definition limitations, see the description of the %define_prj command.
%append <def_list> <def_value>	Adds the <def_value>item into the <def_list>global list and create a list if none has been defined, the item must not be an empty string; the item is added in each case, even if one is contained on the list, the (<def_list>) list is a symbol valid for the project as such, however, no list created by PE can be

Table continues on the next page...

	modified; prior adding the <def_value> all spaces in the beginning / at the end of this value will be removed and macros substituted subsequently; for global symbol definition limitations, see the description of the %define_prj command.
%apploc <def_list> <def_value>	Same as with %append, with a difference of adding into the local list.
%addloc <def_list> <def_value>	Same as with %add, with a difference of adding into the local list.
%define_list <def_list> <def_list>	Creates a list by copying as per the existing one.
%define_list <def_list> %include <file-name>	Creates a list as per the file content (filename a relative path with respect to the PE directory)
	%define_list <def_list> { <def_value> or simple macro-command <def_value> or simple macro-command ...
%define_list}[sorted]	Definition of a local list, each item on a separate line; it is possible to use simple macro commands inside list definition, for example %if, %while, %for, %define, %:, but value of the list should not be accessed and it is not allowed to switch target output file; optional parameter sorted may be used to sort the list items (especially useful to optimize speed of searching items inside large lists)
	%define_list MyList { Item1 Item2 Item3 Item4 %define_list}sorted
%define_list_enabled_components(<def_list>)	Creates list def_list with all enabled components in the project, list contains component names and does not contain target processor; if the list exists, it is replaced
	%define_list_enabled_components(componentList) %for comp from componentList %hint component: %comp %endfor
%for <def_name> from/fromdown <def_list>	Block
%endfor	Block generating actions will match the number of items within the <def_list> list, for each generating action, the <def_value> symbol will achieve a value of a single list item, the <def_value>symbol must not be defined either prior starting or following terminating the command; fromdown will browse the list from behind; in addition, sequence of numbers entered in brackets can also be used in addition to the def_list, for example, [1..5]; number of items in the list is limited to 0xFFFFF to avoid poor response time of the script; see also %for_index and %list_size.
%while (<boolean výraz>)	Block

Table continues on the next page...

%endwhile	While cycle command with a condition at the start; maximal number of loops is limited to 0xFFFFF cycles (to avoid deadlock).

3.2.2.8 Sections

An output into multiple files will be generated from a single driver:

- Initialization of a device (a code for the initialization procedure within the CPU driver),
- Module definition file (header file, *.H, *.DEF),
- Implementation file (*.C, *.MOD),
- Pre-generating headers (and empty bodies - templates) of event procedures into special modules separately for each event,
- debug File designed only for driver debug; may contain anything, should contain a list of all symbols (%ALL_SYMBOLS).

However, only a single (active) file into which the generating action is currently underway will be selected each time. Section separating commands use for switching between the files. These commands are not influenced by a conditional translation. Section generating is optional; the source file can define only certain sections. If the given file cannot access a section, the output will be generated implicitly into the implementation section.

The process is case-sensitive.

Commands defining given sections (the section to end either with a command for switching into another section or file end):

%INITIALIZATION	Generates output in the initialization file.
%ENABLE	Generates output in the initialization file permitting peripheral and component functions.
%INTERFACE	Generates output in the file interface.
%IMPLEMENTATION	Generates output in the implementation file.
%INTERFACE <eventname>	Interface of the event selected.
%IMPLEMENTATION <eventname>	Implementation of the event selected.
%DEBUG	Generates output in a special file <project>\Project_Settings\Component_DEBUG\<component-name>.dbg for debug purposes only. This file is generated only if **Component Development** option is selected. (see **Window > Preferences > Processor Expert > Component Debug Verbose Mode**)

Table continues on the next page...

`%FILE[?][+]["fldr"][<dir>]<filename>`	Generates output into an external file with a specified name (implicitly into the project directory); the optional "?" parameter will cause the command to be ignored if a conditional translation has been disabled; the optional "+" parameter will cause the generated content to be added to the end of the project file (if one exists); for fldr parameter - for description see `%FILE`! Command; `<dir>` - an absolute or relative path with regards to the project directory, in Eclipse it is not allowed to generate file outside Eclipse workspace; the command can only be used within the component/CPU drivers, not for main, event, shared modules etc.; the name of the file to be generated must be unique within the project (a file cannot be generated from multiple components), it is possible to use macros to specify the filename.
`%FILE[?]![<dir>]<filename>`	Denotes the file specified as a part of the application; it is expected that the file exists in the project directory and will not be generated, file addition can be conditioned by a conditional translation; the optional ? parameter will cause the command to be ignored if the conditional translation has been disabled; <dir> - see previous command for description.
`%FILE[?]!"srcdir[;fldr]" [<dir>]<filename>`	Same as previous command, but the file (if not exist in the project) is copied from the srcdir (no macro allowed in this parameter, should be an absolute or relative path with regards to PE system directory, source file must be a plain file on the disk, encoded or packed files are not supported), optional obsolete fldr parameter (no macro allowed in this parameter) will determine into which folder the file is to be filed within project CW Classic (not supported in Eclipse). In case the file is not more generated, PE binary compares source file and project file, if the files are same, the file is removed from project, otherwise user is asked to keep the file as user file or remove from project.
`%FILE[?]!~srcdir[;fldr]~ [<dir>]<filename>`	Same as previous, but macro symbols are replaced inside ~~ characters.
 `%FILE[?]$"srcdir" [<dir>]<filename>`	{*deprecated*}, replaced by `%synchronizeStaticFile()`, same as previous, but additionally the following rules are used to update project file if srcdir is changed compare to previous code generation (for example, changes in the file content are not detected): • If previous source file does not exist, the user is asked if the project file should be updated (for example, replaced by new source file). • If both source files are same: no action. • If the project file is same as previous source file, the project file is updated automatically. • Else user is informed, that user changes were detected in the project file and is asked if it shall be updated.
`%FILE#SAVE <filename>`	Immediately save the file generated as `%FILE` on the disc, `<filename>` must be the name of the file generated by the `%FILE` command.
`%USER_MODULE[!["fldr"]]<filename>`	Generates output into the user module generated; the optional exclamation mark behind the module means that the module will be generated at all times, if not, it will be removed from the project automatically by PE (otherwise, it is only

Table continues on the next page...

	added or enabled within the project and never removed), the optional fldr parameter: see the `%FILE!` command description. The file name is either an absolute path or relative with respect to the generated code directory in PE-Delphi, to the user code directory in PE-Eclipse, the file extension to be in agreement with the one of the code or header file. The command can be used within the component driver or from main, but not from CPU. The name of the file to be generated must be unique within the project (a file cannot be generated from multiple components). If the module exists, it will not be changed similarly as with other user modules.
`%USER_MODULE[!~fldr~]<filename>`	Same as previous, but macro symbols are replaced inside `~~` characters.
`%SUBROUTINE <subroutine> [(param1,param2,...)]`	Defines a sub-program, all subsequent lines will be considered as a sub-program code if not processes (they will be not generated on the output), up to the `%SUBROUTINE_END` command; `<subroutine>` identifies the sub-program (identifier), params are optional sub-program parameters, a comma-separated list of identifiers; for use of the sub-program, see the `%inclSUB` command; the sub-program will be valid until all files have been processed; maximum number of parameters = 16; the `%ifdef` command must not be used for parameter symbols as the symbol parameter value is defined at all times, which may also include an empty string.
`%SUBROUTINE_END`	See the `%SUBROUTINE` command description; the output will be positioned back into the original section.
`%createFileLink(src,dst)`	Creates link do destination file in the Eclipse project; `src` is full file name of the source file; `dst` is relative path and file name to project directory.

```
%createFileLink(%'Dir_PE'lib\PDD\PDD.c,PDD\PDD_Static.c)
%createFileLink(%'Dir_PE'lib\PDD\PDD.h,PDD\PDD_Static.h)
```

`%synchronizeStaticFile(src,dst,[options])`	Synchronizes static file in the project, if the file does not exists it is created, if the file exists it is updated if the src file was changed, if the command is no more generated, the file is removed from project. `src` is full file name of the source file or relative file to Processor Expert directory (either system directory or user components directory). `dst` is relative path and file name to project directory. `options` is comma separated list of the following options in square brackets: • `confirmation` - any modification of the file requires user confirmation.

- autoCreateNew - this option can be used together with confirmation option and creation of the file is not confirmed by the user.
- notCreateNew - the file is not created. It is only updated if exists.

```
%synchronizeStaticFile(Beans\Comp\PDD.c,PDD\PDD.c,[confirmation] )
%synchronizeStaticFile(%'Dir_PE'lib\PDD\PD.h,PDD\PD.h,[confirmation,au toCreateNew])
%synchronizeStaticFile(%'Dir_PE'lib\doc\x.pdf,doc\x.pdf,[notCreateNew] )
```

The command may be applied for the same project file if the source file is same. The options are merged with the following rules:

Requirement #1	*Requirement #2*	*Result*
Confirmation USED	confirmation NOT USED	Confirmation USED
autoCreateNewUSED	autoCreateNewNOTUSED	autoCreateNewNOTUSED
notCreateNew USED	notCreateNewNOTUSED	notCreateNew NOTUSED

`%saveCurrentSectionAs(filename)`

Saves the current section into selected file; filename is full path file-name including extension; file path are create if not exist.

WARNING!: This command works only in component development mode and is designed for development and debugging.

```
%saveCurrentSectionAs(c:\temp\temp.txt)
```

For more details, see Component translation sequence, generating initialization topic.

3.2.2.9 Code identification

`%EMBED_CODE[!][+] <id>, %EMBED_END [msg]`

These commands are designed for code denoting within the main module %IMPLEMENTATION or %USER_MODULE sections; the code denoted this way is to be inserted into the target user module in each generating unless it has been already included.

The text will be inserted into the place to which it is generated from the driver; the place can be found as per the longest from the previous three lines generated (preceding the %EMBED_CODE command) or in case %EMBED_CODE contains the +(plus) option the longest from the three lines following %EMBED_END; if the reference line is not found within the user module, the embed code is not added and an error message is displayed.

The embed code is denoted by an identification text (see `parameter <id>`) that must be a part of the generated text and as per which the existence of the embed code within the user module will be identified, therefore, the identification should be unique; it's recommended to place identification text into the first line of the embed code; if the identification is also found at the last line of the embed code, it is used to detect position of the embed code inside user module; the parameter should be human readable because it can be also displayed in the message in case, user code is not found in the user module;

The optional `!` parameter (exclamation mark) will cause the embed code comparison/substitution in every generating action, the first and last line is found within the user module, the whole content is deleted and replaced by a newly generated embed code; advantages: this cannot be changed by the user, the content can be modified as per generating parameters.

The optional `msg` parameter (text until the end of the line) is displayed in modal dialog in case of generating provided the embed code is inserted or updated into the relevant module; if this parameter is not specified, no message is displayed.

NOTE: *Using this command to affect the main module may cause incompatibility with other PE versions.*

Listing: Example of recommended usage

```
%EMBED_CODE! <specific> initialization
/* Begin of <specific> initialization, DO NOT MODIFY LINES BELOW */

              // ### initialization code here ##

/* End <specific> initialization, DO NOT MODIFY LINES ABOVE */

%EMBED_END <specific> initialization was updated
```

3.2.2.10 Working with component items

```
%set[!] <PropName> <Feature> <Value>
```

Sets the (Feature) feature of the property/method/event in question feature selected from CHG script (identified by the PropName symbol) to a new value (Value). The optional exclamation mark behind the command means that the item value does not need to be set exactly, disabling a report of potential error (serves, for instance, for setting the request for calculation of the PLL clock into PE).

NOTE: The command above will not modify the value of a macro which has been already defined - the macros will be maintained as defined with their original values until the end of script processing.

`<PropName>` is also supported in the following formats: `@InhrItem@Symbol`, where InhrItem stands for identification of the item referring component (for example, an inheritance

item symbol or link-to-component item symbol) or "`@_CPU_@Symbol`" which represents the target processor component; or `@_ProjectOptions_@Symbol` for project options; `@>Component@Symbol` where Component stands for a user name of other component instance within the project; and Symbol stands for the item symbol for the component referred to. The "`%set @..@`" command will only be processed if the component performing the operation has been enabled or the referred component is inherited (such case will be reported as development error). Nevertheless, the inherited component method and event setting will be subject to the limitations described in the `ComponentInheritandSharing.doc` file.

Listing: Example of recommended usage

```
%set IntgPropSymbol Value D:0
%set @_CPU_@Clock Value 1.0
%set @InhrItem@PerphDevice ReqValue DMA0=Channel needed for correct operation of ADC0 in
trigger mode
%set @_CPU_@IntCtrl@PeriphDevice GIC=Peripheral required for proper operation
%set @>%compName@Clock Value 1.0
```

The table listed below contains a list of supported features including their values:

Table 3-1. Supported features

Items	Feature	Value
Method/event	Selection	`always|never|enable|yes|no` **always**= *Generate always, not user-modified,* **never**= *Do not generate, not user-modified,* **yes**= *Generate, user-modified,* **no**= *Do not generate, user-modified,* **enable**= *changes* always/never *values to the value set last time by the user* yes/no, yes/no *values are maintained,* *(`%set` yes/no should not be used at all) (using* `%get` Selection *is not recommended due to frequent omission of a certain setting in the test,* `%getBool` *is recommended instead)*
Method/event	MethodDeclaration, EventDeclaration	Returns declaration (header) of the method; `%set` not supported
Event	EventParamCount	Returns number of event parameters; decimal number
Event	EventParamName[index]	Returns name of event parameter with specified index (index of first parameter is 0)

Table continues on the next page...

Table 3-1. Supported features (continued)

Items	Feature	Value
Event	EventParamType[index]	Returns type of event parameter with specified index (index of first parameter is 0); type is returned in the same format as it is stored in the *.bean* file
Event	EventParamAnsiCType[index]	Returns ANSI-C type of event parameter with specified index (index of first parameter is 0); the following base types are supported: bool, char, uint8_t, int8_t, uint16_t, int16_t, uint32_t, int32_t, uint64_t, int64_t, TPE_Float; if user type is assigned, its name is returned; if the parameter address is passed instead of value, the type is returned with * as suffix
Property	ReadOnly	`yes|no|true|false` specifies if the item is read-only or editable; items set as read-only in the component and template cannot be edited `!no` - will change the value and read-only items in the component or template
Property like groups of items	ReadOnlyALL	Can only be used for `%set`, Any value like `ReadOnly`, The command applies `%set` ReadOnly for all items within the group.
Property	Value	Value as per property • Value format for `%get`: the same as that of the macro. For peripheral-like items, this is a name of device as per database. • Value format for `%set`: matches the one entered in the text field within the inspector. • For integer number (integers) items, entering the value as `<FORM>:<NUMBER>` is recommended, <FORM> stands for H,D,O or B, i.e.: **H** ex, **D** ec, **O** ctal, **B** inary. If the format is not specified, the format as per the current item setting is applied (which leads to errors). • Returns number of list elements for lists (`TListItem` and `TListItemFromFile`), setting to be carried out via #{ number of list elements}. • Returns **0** (generate) or **1** (do not generate) for methods and events.

Table continues on the next page...

Table 3-1. Supported features (continued)

Items	Feature	Value
		• Returns template name for **link** items, user component name for inherited items. • For timing, see the separate description
Boolean property, method, event	Bool	`yes/no` as per item value. Returns only these two identifiers regardless of item symbol generating setting. Useful for method and event setting tests.
Boolean group	BoolValue	Supported for `%get` command only. If a symbol is defined for each item, will return this very symbol; in the reverse case, it will return `yes/no`.
All group property	Expanded	`yes/no`, based on the item status (unpacked/packed)
Property	Error	Text of an error, for `%get`, it will return the text in the column 3
Property	LightError	Sets light error to the property. `%get` not supported. Light errors are supported only if option LightErrorsSupported is listed in the options of the .bean file (see `PEx_Data\Config\XMLSpec \Component.xsd`).
Property	HtmlHint	Gets description of the property in HTML format. %set is not supported.
Property/Method/ Event	IsError	Works only for `%get`, will return yes if the item contains an error due to PE, or no in other cases. Any possible error from CHG/TST is deleted.
Property/Method/ Event	IsEnabled	Returns yes/no depending on whether the item is enabled in the current component setting or not (i.e. will return 'yes' if the item is not version-specific for other CPUs, not contained in a disabled boolean group, or disabled within the enumeration group). The item has to exist.
Property/Method/ Event	Exists?	Returns `yes/no` based on whether the item exists within the component or not
Property/Method/ Event	Warning	Works for `%set` only, sets the message (warning) into the item column 3. Supported in `CHG` and `TS2` scripts only.
Property	Name	Item name - ItemName property (read-only)
Property real or integer or list	MinValue MaxValue	Numerical value **NOTE:** for %get, the real value will be returned with/without 3 decimals
Property list	MaxItem	Returns last element index (from zero)

Table continues on the next page...

Table 3-1. Supported features (continued)

Items	Feature	Value
Property enum or bool or peripheral	Index	Enumerating item index Boolean: **0** for true or **1** for false
Property enum, enumeration-group or bool	IndexFromDefinedValue	Sets item Index as per defined value (i.e. converts the value defined to Index). If the value is not defined, it will set Index to -1 (deletes the values), for enumeration group preserves current value *Function %get(_,IndexFromDefinedValue[def]) returns value -1 if defined value* ***def*** *not found, otherwise index of the corresponding item (decimal integer >=0);*
Property enum or enum-group	DefinedValueFromIndex	`%get( _ ,` **DefinedValueFromIndex** [*index*]) returns defined value for selected index (decimal number from range [0..ItemsCount-1]; `%set` is not supported
Property enum or bool or enum-group or bool-group	IndexError, IndexWarning	Sets an error message (or more specifically warning) to the count value with index specified. **%set** <*symbol*> **IndexError** <*index*><*message*> Does not work for %get. In addition, to display the icons in the value selection, the IconPopup needs to be set within BW. Setting for all errors will be deleted if item type is changed.
Property enum or enum-group	LightIndexError	Sets light error to the value with specified index. `%get` not supported
Property enum or bool or peripheral	ItemsCount	Returns a count of possible enumeration values.
Property enum or bool	TypeSpecName	Type name Item TypeSpecChangeAble property must be set within BW.
Property enum	CustomValueRangeLo, CustomValueRageHi	Sets limits for custom value. `%get` is not supported. It is not allowed to set **CustomValueRangeLo>CustomValue RageHi**.
Property ListItemFromFile	ItemFile	File name (including extension) with item definition, relative path to components root directory). If a list has been assigned to the item, it is deleted before editing. Once edited, the list will be completed to the minimum number of items.
Property ListItemFromFile	ItemFile_ReloadContent	File name (including extension) with item definition, relative path to components root directory).

Table continues on the next page...

Table 3-1. Supported features (continued)

Items	Feature	Value
		If the file is changed, number of items and items configuration is re-loaded from previous configuration (read-only items are not stored for reload). If the file does not exists, Processor Expert displays error message.
Property addr	AddrType	A family of properties in comma-separated brackets, [`external`, `internal`, `RAM`, `ROM`, `FLASH`, `EEPROM`,`code`,`data`]. For a complete list of identifiers, see `InspectorItemFeatures.xls`, `TaddrItem` item, column of controls from CHG script.
Property, method or event	ItemLevel	Setting an item visibility level: `BASIC`, `ADVANCED`, `EXPERT`, "`@ HIDDEN @`". It is recommended using this as least as possible as this may cause the inspector flashing and user confusing in editing (the user will not see the item that is described in the documentation).
Property	Text	Returns the value of the item as can be visible in the item text field in the inspector. For **%set** , this works identically as **%set(_,Value)** . It is recommend using **%get(_,Value)** instead.
Property	AutoSelectedValue	If the property value is *Automatic*, returns auto-selected value (text) of the property in format for UI (empty string if no value auto-selected). Note: usually the same value is also displayed in the third column in case there is no warning; otherwise (if value is not *Automatic*) returns value of the property (text displayed in second column).
Property	IsDefaultValueAutoSelected	Return **yes** if the property is set to <Automatic> and default value is selected without any user requirement for register modification; for pin properties returns **yes** also if no pin is routed; returns no in all other cases. This is designed to detect, if the value is selected based on after reset register configuration without any user requirement for modification. `%set` is not supported.
Property	ExtraText	Writes in the inspector column 3, the write will be enabled only for items displaying implicitly no value in the

Table continues on the next page...

Table 3-1. Supported features (continued)

Items	Feature	Value
		column 3. The text will not disappear in the case of new `CHG` script passing, while for instance errors from `CHG` will be deleted in the case of new passing. If the text is enclosed within`<tt> </tt>`, it will be displayed using Courier font with a constant character width. The following item types will display ExtraText from `CHG` in the column 3 with added specific information on item setting: „CPU frequency".
Property for inheritance	Text Value	Component name Component type
Property link to component	Text Value	Component type Component name
Property link to component	Value	`%set` enables adding the component into the project by setting the new:<component/template name>value; the component will be added once CHG script is closed, with subsequent CHG script restart
Property	InitializationValue	`%get` not supported. `%set` works same as feature Value, but only in case the property was not supported in the component yet during CHG execution; otherwise the command is ignored
Property string-list	Line[index]	Returns/sets the line with a number **index** in the string-list (line numbering from 1). Set will add a new line at the end.
Property import from CPU	ImportSymbols	See Modification of string-list topic for details about possibilities for modification of string-list value using `%set` command. Editing will immediately update the list of items displayed. `%get` *not supported so far.*
Property string-list	Strings	Feature to modify value of the string-list. See Modification of string-list topic for details about possibilities for modification of string-list value using `%set` command.
Property string, real and CPU frequency	RecommendedDropDownValues	Feature to modify drop down list of recommended values for the property. The drop down list is used only to allow easy modification of the value of the property; it is not used for validation. See Modification of string-list topic for details about possibilities for modification of string-list value using `%set` command.

Table continues on the next page...

Table 3-1. Supported features (continued)

Items	Feature	Value
Property	SymbolValue[<symbol>]	Current value of the symbol defined from the item. If the symbol has not been defined, `#undef#` will be returned. Can only be used for %get. The value returned from this feature may be different from value currently defined local symbol (if property value was changed by any previous `%set` command).
TIntrItem	InterruptUsageStrategy	Specifies usage of the interrupt vector by the property. Supported values are: `NOT_USED`, `USED_IF_NOT_ELSEWHERE`, `USED_NOT_EXCLUSIVELY`, `USED_EXCLUSIVELY`
Property	ReqValue	Detailed description in 3.3.2.16
Any property	NumberOfReqValues	`%set` not supported, `%get` returns number of requirements registered to the property using `%set ReqValue` command; return integer decimal number, `0` if there is no requirement; This is used in advanced technique for sharing `TimerUnit_LDD` component
Any property	PE_ItemType	Item type name within PE
Property TListItem and TListItemFromFile	DeleteItemIndex	Deletes an item with specified index from the list, works only for `%set`
Directory like property	BaseDir	Sets a directory to which a selected path is to be set relatively

Example:

```
%set @_ProjectOptions_@MainModuleUpdate Index 4
```

3.2.2.10.1 Modification of string-list

The following values can be passed to `%set` command parameter to affect string-list value:

- `+string` adds a string into the list (if not contained);
- `-string` deletes a string from the list (if contained);
- `[string1,string2,...]` sets a complete list of strings;
- `[]` can be used to assign an empty list.
- def-list assigns a content of the list to the string-list; `def_list` is a identifier of existing list

```
%apploc list One
%apploc list Two
```

```
%apploc list Three

%set STRING_PROPERTY_SYMBOL RecommendedDropDownValues list
```

`%get(<PropName>,<Feature>)`	Returns the feature actually set (`Feature`) of the property/method/event in question (`PropName`), for values see the `%set` command. Can be used for testing the setting following the "`%set`" command as well (for example, `%set MethodName Selection enable`). `<PropName>` can also look like `@InhrItem@Symbol` or `@>Component@Symbol` see the description of the `%set` command.
`%changed(<PropName>)`	Can be used as a condition in `CHG` script files; `<PropName>` is an item symbol; returns `TRUE` if `CHG` script has been called due to editing the given item (user-modified within the inspector); if `<PropName>` equals to `_unKnown_`, then it will return `TRUE` provided the edited item is unknown (`CHG` script will start from other reason); if `<PropName>` equals to `_ANY_`, then `TRUE` is returned if an arbitrary item has been edited.
`%changed(<TGrupItem>)`	Returns `TRUE` to an 'item group' item with no value (not a boolean group), provided any item from the group has been edited.
`%findPropertyByRegExSymbol(<ComponentName>,<RegExprSymbol>)`	Finds a valid property in the component of the specified name, a symbol of which will match the regular expression specified; a property will be valid if it is active and defining symbols - i.e. for instance is not disabled within the boolean group of items; if the component/item does not exist or the regular expression matches multiple items, it will return an error that reads as `#<error description>`.

3.2.2.11 Expressions

Handle real values, for bit operations, operands are rounded to sign 32-bit integers.

```
%:<def_name><unary_operator>(<expression>)[ komentá?]
```

`%:<def_name><unary_operator><number>[ comment ]`	Evaluates the expression and assign the result under the variable <def_name> (only driver-defined symbols can be used, values of the symbols generated by PE cannot be edited); <unary_operator> a <expression> for description, see chapter Identification above, an integer operation is an operation above 32-bit sign type; example: %:a=0; will assign the value 0 to the a symbol, %:a+=1; will add 1 to the a symbol; excess spaces at the command end will be ignored. The comment should be separated by a space and semicolon or by a space and %-.

Table continues on the next page...

%:<def_name>? =<number>,<cislo1>:<number_1>,<cislo2>:<number_2>, ..	Converts the (<number>) value according to the table, if <number>=<cislo1>, then the result will be <number_1>, if <number>=<cislo2>, then the result will be <number_2> etc., the result will be assigned under <def_name> (the symbol will be defined if it is not); failure to find the value in the table will report an error; works for non-negative integers only.

3.2.2.12 Debugging

%ALL_SYMBOLS	Writes all symbols defined (local and global) into the source text, designed for DEBUG file generating.

3.2.2.13 Insertion marks

Insertion marks serve for inserting a code in the case of follow-up generation to the denoted place within the generated code and vice versa, (to generate a code to be used later). Handling the insertion marks is controlled by means of the %THREAD command with subsequent syntax, where <id> is an identifier denoting the thread:

%THREAD <id> CREATE	Creates a new insertion.
%THREAD <id> CREATE_NO_DUPL	Creates a new insertion, in which duplicate lines will be removed automatically.
%THREAD <id> INSERT	Inserts the insert to the point of current code generating action, where each insert can be inserted to a single point.
%THREAD <id> SELECT	Selects the insert as a target file, into which the code is generated (similarly as with the section switching commands, see Sections topic; unlike the section switching one, this command will be subject to conditional translation (switching will be selected only if generating by conditional translation has not been disabled).
%THREAD <id> UNSELECT	Recovers the previous target file into which the code is generated.
%THREAD <id> DESTROY	Releases the insert from the memory, the insert will remain inaccessible, when a new insert with the same identification can be created.

The parameters of the %THREAD command can be combined arbitrarily (but only meaningfully), the processing takes place from the left to the right. For instance:

```
%THREAD thrd CREATE INSERT
%THREAD thrd UNSELECT DESTROY
```

3.2.2.14 %set ReqValue <value>=<reason>

The command creates a request for setting up a value of CPU component item, inherited component, linked component or item of actual component itself, required from the actual component. Value will be set according to the request and will be read only. After the request is cancelled (i.e. component disabled) the initial value of the item will be reset.

It is correct to use several different values (change the value according to the current component setting). For reset is always used the initial value (value used before first ReqValue request). If different components asked for different values, target item will issue an error.

NOTE

Item initial values are stored into the project, after loading the project the required value will be set from `CHG` script.

The value must have the format `<value>=<reason>`, where `<value>` is the requested item value. Generally value displayed in the second column shall be used with the following exceptions:

- value for integer properties must be in format D:num, where num is a decimal number;
- value for time property must be in format H:m:s, for example 13:30:55;
- value for date property must be in format yyyy-mm-dd, for example 2012-01-31;

`<reason>` is the reason for the request shown in the hint. For enumeration items it is also possible to use format `#DefinedValue:<def-value>=<reason>`, where `<def-value>` is value defined from the item (it depends on settings of the property). For boolean items, it is also possible to use format `#Bool:<yes/no>=<reason>`.

The request is cancelled by setting up the value `$=NONE=$`.

Cancelation of all requests from the current component to all items in the target component excluding the `TTmngItem` and `TPrscItem` and `TListItem` items may be done by entering the value `$=CLEAR-ALL=$` into any item of the target component. It is recommended to cancel all the requests on the start of `CHG` script and after that set up all of them again. This is because it is not easy to cancel previous requests (i.e. after switching the peripheral). If there is no need to cancel all the requests (for inherited components, linked

components and the component itself, because all the requests are always updated), value `$=CLEAR ALL NOT APPLICABLE=$` should be used (only suppress the error message from the PE).

All settings and cancelations of requests are done immediately during the command execution, so the interrupt and the following value setup may cause reallocation of the peripheral, invalidate the calculated timing and also malfunctioning of the whole PE project.

Supported is a setup of boolean, enumeration, prescaler type items.

Above that for inherited and linked components items string, priority, integer (value must be in the format D:num), enumeration-group, periphery, signal name (pin), +/- list are supported.

Limitations list (`TListItem`, `TListItemFromFile`) items: ReqValue cannot change value outside range selected by Min/MaxValue of the list item.

3.2.3 Inherited item symbols

This section describes the access to inherited item symbols.

`%@<symb>@<def_name>`	A value of `<def_name>` define-symbol from inherited or shared component, which is referred to by an item with a `<symb>` symbol; in case <symb> is a symbol of an event shared by more descendants, the returned value is unique identified for each descendant (the value depends on component from which the command is executed).
`%[@<symb>@<i>,<def_list>]`	Returns inherited/shared component list item with index < i > (`<symb>` is a symbol of an item that refers to an inherited/ shared component), <def_list> contains name of the list; <i > represents the index, the items are numbered from 1, if the index is out of range, an empty string will be returned (and error displayed in development version), `#undef#` if def_list not defined.

3.2.4 Other macros

The macro can be anywhere within the line; every line is processed from the right to the left. Each macro must start with a per cent mark.

%%	Character %
%<def_name>	Converts to <def_value>.
%'<def_name>	Converts to <def_value>.
%~<def_name>~	Converts to <def_value>, the conversion will be carried out only once, therefore can be applied to a symbol with a value containing the "%" character.
%for_index_1	Index of value currently set in for-variable in the range of 1.. {number of values in the list}; (if from_down is used, the value is decremented in range {number of values in the list}.. 1); the symbol is available only for latest for command.
%for_index_0	The same as with for_index within the range 0..<number of items in the list>-1.
%list_size(<def_name>)	A number of list elements, no spaces between the quotes; <def_name> has to be a variable, it must not be a different macro; if <def_name> is not defined, zero will be returned; I recommend using the %if defined() test instead of %if list_size()>'0'.
%str_length(<par>)	String length, i.e. string character number, <par> is <string> or <def_name>.
%str_pos(<par-sub>,<par-str>)	The first string position within the second string, 0 position starting with 1 if sub-string does not exist and otherwise, case-sensitive, <par> is <string> or <def_name>.
%uppercase(<par>)	Converts the string (all characters) to upper cases, <par> is <string> or <def_name>.
%lowercase(<par>)	Converts the string (all characters) to lower cases, <par> is <string> or <def_name>
%get_index0(<value>,<def_name>)	Returns an index of a list value (numbered from 0); if the list does not contain the value, -1 will be returned; the list has to be defined.
%get_index1(<value>,<def_name>)	Returns an index of a list value (numbered from 1); if the list does not contain the value, 1will be returned; the list has to be defined.
%short_path(xx)	Converts the absolute path (or file name as well) to a short path for 16-bit applications. The parameter is a macro or text within parentheses. The path (or file) requested has to exist.
%file_exist(xx)	{*deprecated*} %prj/pex_file_exists() should be used instead; returns yes or no depending on existence of the specified file on the disk; the xx parameter should be the absolute path (in Eclipse it must be Processor Expert system subdirectory) and file name, or a relative path with respect to the project directory; the function will not consider whether the file is created in course of the generation process (all generated files are saved in the target directory only *after* all files have been generated successfully)
%prj_file_exists(relpath)	Returns yes or no depending on existence of the specified file on the disk; the relpath parameter should be relative path with respect to the project directory; the function will not consider whether the file is created in course of the generation process (all generated files are saved in the target directory only *after* all files have been generated successfully).

Table continues on the next page...

%pex_file_exists(relpath)	Returns yes or no depending on existence of the specified file *on the disk*; the relpath parameter should be relative path from Processor Expert data directory (system directory or user components directory); the function ignores files located in packages, returns only files on the disk.
%sys_file_exists(relpath)	Returns yes or no depending on existence of the specified system file *on the disk or in the package*; the relpath parameter should be relative path from Processor Expert data directory (system directory or user components directory).
%get_file_list(<filetype>,<def_list>,<filemask>)	Retrieves the list of files on the disk (optionally also sub-directories) with specified mask on the disk and returns number of found files (number of items in the list). <filetype> can be FILES to retrieve list of files or FILES_DIRS to retrieve also list of sub-directories. <def_list> is identifier of new list, that shall be created by the function, the symbol may not exist, the list will be defined only if at least one file is found and will contain absolute paths. <filemask> is absolute path and specification of file-mask, wildcards are not supported in path, however they may be used in file-mask (use * to list all files), file-mask may additionally specify *\ prefix to proceed all sub-directories non-recursively or **\ prefix to proceed all sub-directories recursively; the file-mask may be finished by \. to return names of the sub-directories; the function provides direct file-access to the files on the disk, it does not support files in packages, and does not support Eclipse file-system, so it shall not be used to search files in the project. Example: %:num=%get_file_list(FILES,FileList,c:\Users*.pe), %:num=%get_file_list(FILES,FileList,c:\Users**.pe), %:num=%get_file_list(FILES,FileList,c:\Users***.pe), %:num=%get_file_list(FILES_DIRS,FileList,c:\Users**\.)
%get_prj_file_list(<filetype> <def_list> <filemask>)	Same as get_file_list, but works with Eclipse project files (including hidden files – name starting with "."); the function works with Eclipse directory separator "/" independent on operating system;
%get_prj_file_location(<fileName>)	returns source file (absolute path) for linked project file; returns empty string if the project file not found or it is not linked;
%html_link(<xx>)	Converts the directory to an HTML link, the double characters (< and >) to enclose the parameter will be necessary due to the directory name containing a bracket character.
%sinus(<value>)	Calculates sinus.
%round(<value>[,<prec>])	Rounds a real number to specified number of decimals (range 0-9), if the number of decimals is not specified, rounding to integers will be performed; serves for formatting - i.e. results may include a number with specified number of decimals even in the case there are nulls at the end, for example, %round(1.5)=2, %round(1.5,2)=1.50.
%msg(<id>,<text>)	Displays a dialogue with the relevant text (in which the \n sequence will be replaced by the end of line); id presents one of the following dialogue types (buttons in brackets are displayed behind each description): INFO - Information (OK), YN_CNFRM - Prompt/confirmation (Yes, No), YNC_CNFRM -

Table continues on the next page...

	Prompt/confirmation (Yes, No, Cancel), YN_CNFRM_N - As with YN_CNFRM, but the default button will be No, YNC_CNFRM_N - As with YNC_CNFRM, but the default button will be No, WARN - Warning (OK), ERROR - Error (OK), C_ERROR - Error (Cancel), INTERNAL_ERROR - PE internal error (OK); the function will result in a button, which was pressed by the user or Cancel.
%makeIdentifier(<text>)	A macro function to convert the string characters to identifiers, replacing unsupported characters by an underscore; have on mind: the function does not check if the first character is non numeric.
%getCurrentTime()	Returns current time and date in the format as per Windows settings.
%toAscii(<number>)	Converts an 8-bit positive integer to an ASCII character; if that number is outside the range of characters that can be displayed, a HEX value in the format XX will be returned.
%EXPR(<expression>)	Evaluates the expression and returns its value.
%createProjectItem(<itemtype>,<params>[,EnableInAllConfigurations])	Creates item for projects of the specified types; <itemtype> can be: • Configuration - creates configuration, <params> is its name. • Bean - creates a component or template of the specified type <params>; local template supported as well; for inserting a specific CPU variant, a parameter in the format as follows can be entered: <CPU-component>=<CPU-variant> • Bean_autoconnect - creates a component or template of the type specified <params>, performing auto-connect to the target CPU. optional parameter EnableInAllConfigurations can be used to add component enabled in all configurations (by default it is enabled in current configuration only) • Whichever is the case, defines the CreatedProjectItemName symbol with a value presenting the name of a newly created object.
%addAllPeripheralInitComponents()	Adds peripheral initialization components for all peripherals on the selected target processor; if supported adds also PinSettings component; returns number of added components; the function is designed to be used by new project wizard.
%getTargetFileName()	Returns file name with extension (without directory) of currently selected output file; If no output file selected or event or %INITIALIZATION or %THREAD is selected, returns empty string.
%convertPathToOsSpecificFormat(<path>)	Converts path to OS-specific directories delimiters, <path> is <string> or <def_name>. Processor Expert uses backslash as default path delimiter, this function can format output path to be compatible with operating system ,where Processor Expert is running; this function is supported only in PE Java.
%convertListToString(<def_name>,<string>)	Converts list to string, first parameter is name of existing list, second parameter is string used as a delimiter of list values (can be empty string); all list items are added into string

Table continues on the next page...

	delimited by specified delimiter; the function can be used to generate list content to output or pass the list as `%launchExt` parameter.
`%[<i>,<def_list>]`	Returns a list item no. i, where items are numbered from 1, if the index is out of range, an empty string will be returned, `def_list` has to exist.
`%#<srcf><dstf>[-]<number>`	Converts an integer (`-0x7FFFFFFF..0xFFFFFFFF`) from a specified into the requested format:
`<srcf>`	"(*if not specified*) - decimal number,
	' **d** ' - decimal number,
	' **2** ' - binary number,
	' **L** ' - non-sign 32-bit number,
	' **H** ' - hex number,
`<dstf>`	' **h** ' - high-level language format
	' **d** ' - decimal number
	' **a** ' - assembler format - data
	' **aa** ' - assembler format - address
	' **ab** ' - assembler format - binary data
	' **B** ' - binary number (without a prefix and extension)
`%#R<number>`	Real number formatting.

Serves for outputs to help, linker, maker and debug files and comments:

`%#b<number>`	Converts a number from a decimal format to a hex form ?? (without a prefix and extension) , to generate constants, prefer `%#h` or `%#a`.
`%#w<number>`	Converts a number from a decimal into a hex format (4 digits without a prefix and extension), to generate constants, prefer `%#h` or `%#a`.
`%#l<number>`	Converts a number from a decimal into a hex format (8 digits without a prefix and extension), to generate constants, prefer `%#h` or `%#a`. .

3.3 Component translation sequence, generating initialization

First, a source code is generated (only the sections as follows: interface/implementation, help, linker and maker) from a SRC file named "`Main.src`" that should generate (entire) main module - namely the interface and implementation part (generating into other sections is optional).

Subsequently, modules for each device are generated; these modules can add the code into the joint initialization procedure (all of them will add a code only). The initialization procedure is a part of the processor module. Further, the module as such will be generated from their SRC file (header as well as implementation) that will implement the functions requested by the user. If handlers for certain events are required by the user as well, a template (again, header and implementation) is generated for writing the code for these events by the user. The template will be generated into the shared modules (multiple drivers can share a single module - for module name, see `%EventModule`).

Afterwards, shared modules are generated from the SharedModules global list.

Finally, the processor module is generated. At the same time, this module represents an initialization module; an initialization and enable codes that had been successively generated by each component are added to the end of the module. This means that the resulting processor (implementation) module will be composed step by step from the following sections: `%IMPLEMENTATION` from the processor, `%INITIALIZATION` from all components, `%ENABLE` from the processor, `%ENABLE` from all components, `%INITIALIZATION` from the processor. In addition, termination of the initialization code must be generated by the processor driver (for example, `END` command), into the initialization section (`%INITIALIZATION`), i.e. to the end of the initialization and enable codes. Similarly, the same will go for assembler initialization, except for `%ENABLE`, i.e. the section sequence will be as follows `%ASSEMBLER` and all `%INITIALIZATION` asm.

Further, the processor module usually also generates initialization code for interrupt vectors.

The processor driver can terminate generating files for `LINKER` and `MAKER`, as well as the project for the target compiler.

Once drivers for each device and processor module have been generated, files containing templates for each event are completed; these modules will be created by merging codes from relevant drivers and generating the "event.src" file and adding its "interface" and "implementation" sections to the beginning and "interface end" and "implementation end" sections to the end of the generated source code from all modules.

COMPONENT.SRC

`INTERFACE`	Header of the device module.
`IMPLEMENTATION`	Implementation of the device module.
`INTERFACE <event>`	Generates into the header of the events module.
`IMPLEMENTATION <event>`	Generates into the implementation of the events module.
`INITIALIZATION`	Generates into the implementation of the processor module.
`ENABLE`	Generates into the implementation of the processor module.
`LINKER`	Generates into the *linker* file.

Table continues on the next page...

MAKER	Generates into the *maker* file.

3.4 Generated code format: requirements

Processor Expert defines certain additional conventions for the generated code that allow for finding implementation of driver method from the environment (using a double-click on the method/event on the Project Panel), this is performed via searching for method/ event name in the comments maintain the user code in the event implementation. Additionally, the following actions are applied to user (event) module after successful code generation,

- *Module and event name renaming:*
  - Execute for all renamed modules and events - Change all names to internal identification
  - Execute for all renamed modules and events- Change all internal identifications to new names
- *Deleting imports not generated anymore.* For all imports within the user module,
  - If not exists on the list of all generated modules within the current project, then
  - If generated the last time, then delete it
- *Adding new imports*. For all newly generated imports,
  - If not exists within the user module, then add it behind the last import
- *Adding new events.* For all newly generated events,
  - If not exists within the user module, then add it to the module end (before MODULE END)
- *Deleting events not used anymore*.
  - On option, if the event does not contain any user code

3.5 Limitations of code generation

There is limit of total lines for one script to avoid recursive %include, which could cause abnormal memory allocation and invalid operation. Currently maximal number of lines is 768 000 lines.

Chapter 4
TST script for component testing

The files are stored in the driver directory on the same location as drv files, but they possess the tst extension; they are processed by PE macro processor using src file, with the only difference of the `DriverExtension` macro having `tst` instead of `drv` value. The existence of the TST script file is not necessary (unlike the driver). The output from this file is not expected and is not saved anywhere, only errors, warnings and hints or possibly setting by the `%set` command will be accepted. Prior processing the component TST script file, `main.tst` will be processed (if exists). In the course of tst processing, defining global symbols (except for the `main.tst` driver) will not be permitted - any possible symbols will be defined only locally, when the symbols from the main.TST script file will be available for component `TST` as well.

Warning

If a TST script file exists for the driver, the driver will not generate the %warning and %hint messages (these are expected to have been generated by means of the corresponding TST script file).

`TS2 script file` will be in most times started only after `tst`.

Chapter 5
Component scripts

The following are supported component scripts:

- CHG script - validation of component settings; executed even if the component contains errors
- TST script - validation of component settings; executed after CHG script only if the component does not contain any error
- TS2 script - inter-component validations; executed after TST script only if the component does not contain any error
- DRV script - component code generation script (for example,. executed during code generation)

5.1 CHG script: setting control script

The scripts serve for HW-independent as well as HW-dependant test of setting of the given component or compiler with the possibility of changing the setting (that is setting for specific component properties, methods and events or compiler setting). The scripts are located in the component directory (the same as with the `*.bean` file); the extension reads as chg; for types and naming of scripts, see CHG script types topic. These files are processed by PE in the same way as that for the drivers, starting them directly without using the src files; in addition, these files can employ the `%set` command. The `DriverExtension` macro will have the `chg` value. Unlike the src file, the existence of the CHG script file is not required. The output from this script will not be saved anywhere, only errors, warnings and hints will be displayed and item setting will be executed using the `%set` command. Other non empty line generated to output is considered as wrong CHG script. Generating errors directly for the corresponding item using the " `%set Property Error ErrorMessage`" command is recommended.

NOTE

Within the CHG script file processing, the symbols defined will not be changed, even if the item value is modified by the `%set` command. The current item setting value (for example, after the `%setMethod Selection enable` command) can be tested using the `%get` command. Instead of testing the value of the corresponding symbol, only use of the `%get` command is recommended.

The CHG script file will be processed each time the component setting is modified regardless of correctness of the setting or if a target processor had been selected or not. During the process above, some symbols may not be defined (no symbol will be guaranteed). Within the CHG script file, work will be possible only with the symbols valid for the given component, but not with global symbols. Inserting the component into the project will open CHG script with the defined `BeanInitialization` symbol (which does not apply for project loading); during project loading CHG file is executed after component is loaded with `symbol _SpecialBeanCompatibilityInitialization` defined; creation of inherited component will invoke CHG script of parent component with symbol `_BeanInInheritedItemInitialization` and its value is symbol of property, that contains new inherited component.

In addition, during execution of `CHG` script will be defined the symbol `CHG_BeanIsEnabled` with yes or no values provided information if the component is enabled.

5.2 CHG script types

The common `CHG` script has the same name as the component/compiler. A script specific only for a certain processor family must be enlarged by `_<family>` suffix in file name; this script will be executed after the common script (of course only if the target processor belongs to the corresponding family). Finally, a script may exist with a name extended by `_ZZ_finish`; such script will be run in the end.

All the `CHG` scripts above are optional; however, it is recommended that at least common `CHG` script exists. If common `CHG` does not exists, none of these described above will be launched.

Chapter 6
TS2 script for component interdependence testing

TS2 scripts have been designed to allow for the following:

- Component interdependence testing
- Definition of global symbols for generating

TS2 script will be processed in the same way as TST script, with the only differences as follows:

- `TS2` script can define global symbols that will remain defined for generating codes
- `TS2` script will always be processed for all components at the same time
- `TS2` script can set errors to items of arbitrary components by the `%set @>Component@Item Error` command
- `TS2` script can set an extra text to items of arbitrary components by the `%set @>Component@Item ExtraText` command
- `TS2` script can influence the sequence of their processing as follows: at the start of the `TS2` script (the file used from `SRC`) the first line (or lines) in the format like `% AFTER_BEAN:component`, where component stands for the name of the component, the TS2 script file of which is to be processed prior this component. Init components: for each family, a component name can be entered in the format as follows: `Init_<name>_*`. The sequence is determined statically, i.e. without conditional translation. The sequence determined this way is preferred to that determined by a reference to a different component instance (inherited or shared components).

NOTE

Any modification will need project reloading.

Chapter 7
CDB

CDB file contains information about supported components for the selected processor and selected peripheral. The CDB files are created/updated automatically by Processor Expert, after new component is installed.The CDB file is stored within the directory of the processor component.

7.1 readyMASK

readyMASK is used in the component code-generation script (.drv) to identify supported derivatives and supported peripherals by the component and the component script.

```
%- readyMASK families={F}[,..] [CPUDBversion>={V.VV.VVV}]
peripherals={P}[,{P},...] [special_reg={SR}] [not_special_reg={SR}]
[special_CPU_reg={SR}] [numOfPrphInstances>={NumInst}]

%- notreadyMASK families={F} [subfamily={S}]
[cpu_components={C}[,...]] peripherals={P}[,...]

%- notreadyMASK families={F} [subfamily={S}]
[cpu_components={C}[,...]] peripherals={P}[,...]

%- notreadyMASK families={F} [subfamily={S}]
[cpu_components={C}[,...]] peripherals={P}[,...]
```

Where, {F} is identifier of Processor family from CPUDB; {S} is identifier of Processor subfamily from CPUDB;

{C} is identifier/mask of Processor component, mask can be specified using * and ? wildcards;

{V.VV.VVV} is version of CPUDB database for compare, it is not necessary to specify complete version number, for example CPUDBversion>=3 can be used as well;

{P} is name of the peripheral from CPUDB, that is required for functionality of the component; it is not allowed to use peripheral names, that are not supported on any processor,

{SR} is an identifier of special register (function), that is required (or conflicting) to be assigned to the peripherals (for example: it must be used for EventCounter component, which can allocate only counter, that contains external clock pin - so peripheral name is not sufficient, it is necessary to specify extra functionality); if both special_reg and not_special_reg are used, so the peripheral must contain the first special register and may not contain the second special register; special_CPU_reg is used to specify required register in CPU peripheral.

{NumInst} decimal integer number specifying minimal number of peripheral instances; user-case: should be used for Byte2/3/4IO components that require at least 2/3/4 port instances

readyMASK is used to specify list of Processor components and peripherals, which are supported by the component, while notreadyMASK is used to specify exceptions (what is not supported yet). It is expected, that notreadyMASK will be used only as an exceptional or temporary solution.

Listing: Example

```
%- readyMASK families=HCS08,ColdFireV1,RS08 CPUDBversion>=3
peripherals=PTA,PTB,PTC
%- readyMASK families=RS08 peripherals=TPM
special_reg=PE_PRPHREQ_EVENTCOUNTER
```

Chapter 8
External libraries

The PE macro-processor language allows invoking of code from external dynamically loaded libraries (external objects). This chapter describes how to implement those libraries and provides description of their API. Supported are MS Windows DLL libraries via `%launchDLL` command and on Eclipse pure Java PE-service platform `%launchExt` command is also available extending `%launchDLL` functionality to Java class libraries, Java class libraries stored in JAR files and shared objects on Linux systems.

This chapter describes required interface of external shared libraries so PE macro language is able to use them.

To make shared libraries functions accessible by PE macro language, unified API has to be used for them. Required API for Java libraries (`.class` and `.jar` files) and OS shared libraries (dynamic-link libraries and shared objects) differs as Java libraries are accessible directly from PE Eclipse implementation contrary to OS shared libraries which are accessible only through Java Native Interface.

The topics covered here are as follows:

8.1 OS shared libraries API

Following API should be used in MS Windows dynamic-link libraries (`.dll` files) and Linux shared objects (`.so` files) accessed by `%launchExt` macro language command (in case of dll libraries it is also possible to use `%launchDLL`).

Implmenetation node:*Interface between PE and external library is made through stub library, which is implemented for both MS Windows and Linux platform as part of PE in Eclipse. This library manages sub-layer between Java environment and OS environment using Java Native Interface.*

PE is supporting two C language function calling conventions: "regparm" and "stdcall". Distinction between them is determined by `_STD` suffix in library function name, which usage is obligatory for functions implemented with stdcall convention. However, there is no explicit specification of used calling convention in `%launchDLL/%launchExt` command and library is searched in following sequence:

1. Function `<function-name>` is searched as first.
2. If `<function-name>` is not found, `<function-name>_STD` is searched.

This means that:

- every external library function has to have an unique name - `_STD` suffix can't be used to differentiate two functions names,
- name of function implementing `stdcall` convention have to end with `_STD` suffix and
- name of function implementing `regparm` convention can not end with `_STD` suffix.

Functions declaration:

```
char * __stdcall <function-name>_STD (const char *params, char
  **macroCmds, char **defineSymbols);

 char * __regparm(3) <function-name> (const char *params, char
  **macroCmds, char **defineSymbols);
```

`params` - string containing parameters passed to the function

`macroCmds` - pointer to string used to pass PE macro language commands from function. After processing `%launchExt/%launchDll` command, it can be processed additional macro commands from `macroCmds` content. Multiple commands stored in `macroCmds` has to be separated with `\n` or `\r\n` sequence. See Macro commands topic for list of supported macro commands.

`defineSymbol` - pointer to string - deprecated, used only for backward compatibility

`returns` - string with lines directly sent to the output of the script

This is the basic declaration of API which should be used. Particular declaration used in external library implementation may differ depending on compiler used to build the shared library (`.DLL`/ `.SO`) - supported attribute declaration may slightly differ.

Listing: Example

```
char local_buffer[1000];
char* __attribute__((stdcall)) returnInputSTD_STD (char *params, char
**MacroCmds, char **DefineSymbols)
{
```

```
    sprintf(local_buffer, params);

    *MacroCmds = local_buffer;

    return local_buffer;
}
char* __attribute__((regparm(3))) returnInput (char *params, char
**MacroCmds, char **DefineSymbols)
{
    sprintf(local_buffer, params);
    *MacroCmds = local_buffer;
    return local_buffer;
}
```

Example of functions just returning string passed in params input parameter is showed above using GCC attribute declaration, one using " `regparm`" calling convention and one using " `stdcall`" convention.

Next example shows possible declaration of same functions when using Microsoft

Listing: Example - 2

```
 __declspec (dllexport) keyword used to export these functions:
__declspec (dllexport) char* __attribute__((stdcall))
returnInputSTD_STD (char *params, char **MacroCmds, char
**DefineSymbols)
{
    sprintf(local_buffer, params);
    *MacroCmds = local_buffer;
    return local_buffer;
}
__declspec (dllexport) char* __attribute__((regparm(3))) returnInput
(char *params, char **MacroCmds, char **DefineSymbols)
{
    sprintf(local_buffer, params);
    *MacroCmds = local_buffer;
    return local_buffer;
}
```

8.2 Java libraries API

External libraries functions stored in `.class` or `.jar` files API:

```
String[][]<function_name> (Object component, String params);
```

`component` - interface to processor expert component context `com.processorexpert.core.service.api.IPExComponentAPI; null` if the calling script is not running for any component;

`params` - string containing parameters passed to the function

`returns` - String array of two elements - first element contains array of string that will be generated to script output, second element contains array of macro command strings (see Macro commands topic for list of available commands; `%include`, `%inclSUB`, `%launchExt` and `%launchDLL` are not supported).

Listing: Example

```
class LaunchExtObjectTestClass {
  public String[][] testMethod(Object component, String params) {

    String[][] result = new String[2][2];

    //to generate on the script output

    if (component == null) {

      result[0][0] = null;

      result[0][1] = params;

    } else {

      result[0][0] = component.toString();

      result[0][1] = params + component.toString();

    }

    //macro-commands to execute

    result[1][0] = "%set ItmSymbol1 Text YYY1";

    result[1][1] = "%set ItmSymbol2 Text YYY2";

    //

    return result;

  }

}

%launchExt LaunchExtObjectTestClass.class,testMethod,PARAM
```

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Document Number CWPEXMLREF
Revision 10.6, 02/2014