

## 13 Implementation Integration

### 13.1 Compiler Abstraction (Compiler.h)

The compiler abstraction specifies definitions and macros for the abstraction of compiler specific keywords used for addressing data and code (pointers, variables, function etc.) within declarations and definitions.

The file **Compiler\_Cfg.h** contains the module / component specific parameters (ptrclass and memclass) that are passed to the macros defined in **Compiler.h**.

#### 13.1.1 Definitions

Define	Description
<code>AUTOMATIC</code>	Empty definition, used for the declaration of local pointers.
<code>TYPEDEF</code>	Empty definition, used within type definitions, where no memory qualifier can be specified.
<code>NULL_PTR</code>	NULL Pointer <code>((void *)0)</code>
<code>INLINE</code>	Define for abstraction of the keyword “inline”.
<code>LOCAL_INLINE</code>	Define for abstraction of the keyword “inline” in functions with “static” scope.

Table 209 Compiler Defines

#### 13.1.2 Memory Classes and Pointer Classes

In the following table the `<PREFIX>` is derived from the Software Component's `shortName`.

Memory type	Syntax of memory class (memclass) and pointer class (ptrclass) macro parameter	To be used for...
Code	<code>&lt;PREFIX&gt;_CODE</code>	code
Constants	<code>&lt;PREFIX&gt;_CONST</code>	global or static constants
Pointer	<code>&lt;PREFIX&gt;_APPL_DATA</code>	references on application data (expected to be in RAM or ROM) passed via API
Pointer	<code>&lt;PREFIX&gt;_APPL_CONST</code>	references on application constants (expected to be certainly in ROM, for instance pointer of Init-function) passed via API
Pointer	<code>&lt;PREFIX&gt;_APPL_CODE</code>	references on application functions. (e.g. call back function pointers)
Variables	<code>&lt;PREFIX&gt;_CALLOUT_CODE</code>	references on application functions. (e.g. callout function pointers)
Variables	<code>&lt;PREFIX&gt;_VAR_NOINIT</code>	all global or static variables that are never initialized
Variables	<code>&lt;PREFIX&gt;_VAR_POWER_ON_INIT</code>	all global or static variables that are initialized only after power on reset
Variables	<code>&lt;PREFIX&gt;_VAR_FAST</code>	all global or static variables that have at least one of the following properties: <ul style="list-style-type: none"> <li>accessed bitwise</li> <li>frequently used</li> <li>high number of accesses in source code</li> </ul>
Variables	<code>&lt;PREFIX&gt;_VAR</code>	global or static variables that are initialized after every reset.
Variables	<code>AUTOMATIC</code>	local non static variables
Type Definitions	<code>TYPEDEF</code>	type definitions where no memory qualifier can be specified.

## 13 Implementation Integration

Table 210 Memory Classes and Pointer Classes

### 13.1.3 Macros for Functions, Pointers, Constants and Variables

These macros ensure correct syntax of definitions and declarations independent of any specific compiler.

Macro	Macro usage (define)	Short description and example
FUNC	FUNC( rettype, memclass )	<b>Function declaration and definition</b> FUNC( void, SwctShortName_CODE ) RunnableCode( void );
FUNC_P2CONST	FUNC_P2CONST( rettype, ptrclass, memclass )	<b>Function returning a pointer to a constant</b> FUNC_P2CONST( uint16, <PREFIX>_PBCFG, SwctShortName_CODE ) ExampleFunction( void );
FUNC_P2VAR	FUNC_P2VAR( rettype, ptrclass, memclass )	<b>Function returning a pointer to a variable</b> FUNC_P2VAR( uint16, <PREFIX>_PBCFG, SwctShortName_CODE ) ExampleFunction( void );
P2VAR	P2VAR( ptrtype, memclass, ptrclass )	<b>Pointer to variable</b> P2VAR( uint8, SPI_VAR_FAST, SPI_APPL_DATA ) Spi_FastPointerToApplData;
P2CONST	P2CONST( ptrtype, memclass, ptrclass )	<b>Pointer to constant</b> P2CONST( Eep_ConfigType, EEP_VAR, EEP_APPL_CONST ) Eep_ConfigurationPtr;
CONSTP2VAR	CONSTP2VAR( ptrtype, memclass, ptrclass )	<b>Constant pointer to variable</b> CONSTP2VAR( uint8, NVM_VAR, NVM_APPL_DATA ) Nvm_PointerToRamMirror = Appl_RamMirror;
CONSTP2CONST	CONSTP2CONST( ptrtype, memclass, ptrclass )	<b>Constant pointer to constant</b> CONSTP2CONST( Can_PBCfgType, CAN_CONST, CAN_PBCFG_CONST ) Can_PostbuildCfgData = CanPBCfgDataSet;
P2FUNC	P2FUNC( rettype, ptrclass, fctname )	<b>Pointer to function</b> typedef P2FUNC( void, NVM_APPL_CODE, Nvm_CbkEncPtrType ) (void);
CONST	CONST( consttype, memclass )	<b>Constant</b> CONST( uint8, NVM_CONST ) Nvm_ConfigurationData;
VAR	VAR( vartype, memclass )	<b>Variable</b> VAR( uint8, AUTOMATIC ) Nvm_VeryFrequentlyUsedState;
<b>vartype:</b> <b>consttype:</b> <b>rettype:</b> <b>ptrtype:</b>	type of the variable type of the constant return type of the function type of the referenced variable/constant	<b>fctname:</b> function name respectively name of the defined type <b>memclass:</b> classification of the function/pointer/constant/variable itself <b>ptrclass:</b> classification of the pointer's distance →13.1.2

### 13.2 Memory Mapping (MemMap.h)

In order to prevent unnecessary memory gaps (unused space in RAM), variables of different size (8, 16 and 32 bit) are mapped to specific memory sections depending on their size.

Variables that do not need to be initialized after power-on reset can be mapped to a RAM section that is not initialized after a reset.

Variables that are accessed via bit masks can be located within a RAM section that allows for bit manipulation instructions of the compiler (“Near Page” or “Zero Page”).

When using external flash memory, modules with functions that are called very often can be mapped to the internal flash memory for faster access. Modules with functions that are called rarely or that have lower performance requirements can be mapped to external flash memory.

Internal variables can be mapped into protected memory for memory protection, buffers for data exchange are mapped into unprotected memory.

In order to support Partitions (→17) an additional separation of the module variables into different memory areas (of the Partitions) is supported.

All memory allocated by the RTE is wrapped in segment declarations defined in the specification of Memory Mapping using RTE as the <MSN> (Module Short Name).

#### 13.2.1 Include Strategy of MemMap.h

- Basic Software Modules include **MemMap.h**
- Software Components include **<SWCT>\_MemMap.h**  
(SWCT = shortName of the SwComponentType)

#### 13.2.2 ALIGNMENT

The alignment is used for the Memory Allocation Keywords →13.2.4.

<ALIGNMENT>	Used for...
BOOLEAN	variables and constants of size 1 bit
8BIT	variables and constants that have to be aligned to 8 bit.
16BIT	variables and constants that have to be aligned to 16 bit.
32BIT	variables and constants that have to be aligned to 32 bit.
UNSPECIFIED	variables, constants, structure, array and unions when SIZE (alignment) does not fit the criteria of 8, 16 or 32 bit.

Table 211 ALIGNMENT

#### 13.2.3 INIT\_POLICY

The initialization policy is used for the Memory Allocation Keywords →13.2.4.

<INIT_POLICY>	Used for variables that are ...
NO_INIT	never cleared and never initialized.
CLEARED ZERO_INIT <sup>(1)</sup>	cleared to zero after every reset.
POWER_ON_CLEARED	cleared to zero only after power on reset.
INIT	initialized with <code>initValue</code> (→5.18) after every reset.
POWER_ON_INIT	initialized with <code>initValue</code> (→5.18) only after power on reset.

Table 212 INIT\_POLICY

<sup>(1)</sup> This is actually not specified in the AUTOSAR specifications but often used by different vendors.

### 13.2.4 Memory Allocation Keywords (`_START_SEC_` and `_STOP_SEC_`)

For each Software Component, the declaration and definition of code, variables and constants need to be “wrapped” using the `_START_SEC_` and `_STOP_SEC_` mechanism:

```
<PREFIX> _START_SEC_ <NAME> <MEMORY_ALLOCATION_KEYWORDS>
#include <MemMap.h>

/* Here comes the declaration and definition of code, variables and constants */
/* ... */

<PREFIX> _STOP_SEC_ <NAME> <MEMORY_ALLOCATION_KEYWORDS>
#include <MemMap.h>
```

`<PREFIX>` is the `shortName` of the `SwComponentType` (case sensitive)

`<NAME>` is the `shortName` of a Software Component Description (case sensitive) if the `MemorySection` (10.8) has no `symbol` attribute defined. Otherwise `symbol` is used for `<NAME>`.

`<MEMORY_ALLOCATION_KEYWORDS>` is used according to Memory Allocation Keywords according to the following table. Example:

```
#define MySwc _START_SEC_CODE          /* Definition of start symbol for module memory section */
#include "MySwc_MemMap.h"                /* Include of the memory mapping header file */

FUNC(void, MySwc_CODE) Run1( void );    /* Declaration / definition of code, variables or constants
                                         belonging to the specified section */
#define MySwc _STOP_SEC_CODE          /* Definition of stop symbol for module memory section */
#include "MySwc_MemMap.h"                /* Include of the memory mapping header file */
```

For code which is invariably implemented as inline function the wrapping with Memory Allocation Keywords is not required.

Memory Section Type	Syntax of Memory Allocation Keyword
VAR	<pre>&lt;PREFIX&gt;_START_SEC_VAR &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_VAR &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <p>all global or static variables.</p>
VAR_FAST	<pre>&lt;PREFIX&gt;_START_SEC_VAR_FAST &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_VAR_FAST &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <p>Global or static variables that have at least one of the following properties (to save code and runtime):</p> <ul style="list-style-type: none"> <li>• accessed bitwise</li> <li>• frequently used</li> <li>• high number of accesses in source code</li> </ul>
VAR_SLOW	<pre>&lt;PREFIX&gt;_START_SEC_VAR_SLOW &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_VAR_SLOW &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <p>all infrequently accessed global or static variables.</p>
INTERNAL_VAR	<pre>&lt;PREFIX&gt;_START_SEC_INTERNAL_VAR &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_INTERNAL_VAR &lt;INIT_POLICY&gt; &lt;ALIGNMENT&gt;</pre> <p>global or static variables those are accessible from a calibration tool.</p>
VAR_SAVED_ZONE	<pre>&lt;PREFIX&gt;_START_SEC_VAR_SAVED_ZONE&lt;X&gt; &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_VAR_SAVED_ZONE&lt;X&gt; &lt;ALIGNMENT&gt;</pre> <p>RAM buffers of variables saved in nonvolatile memory.</p>
CONST_SAVED_RECOVERY_ZONE	<pre>&lt;PREFIX&gt;_START_SEC_CONST_SAVED_RECOVERY_ZONE&lt;X&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CONST_SAVED_RECOVERY_ZONE&lt;X&gt;</pre> <p>ROM buffers of variables saved in nonvolatile memory.</p>
CONST	<pre>&lt;PREFIX&gt;_START_SEC_CONST &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CONST &lt;ALIGNMENT&gt;</pre> <p>global or static constants.</p>
CALIB	<pre>&lt;PREFIX&gt;_START_SEC_CALIB &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CALIB &lt;ALIGNMENT&gt;</pre> <p>global or static constants.</p>
CARTO	<pre>&lt;PREFIX&gt;_START_SEC_CARTO &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CARTO &lt;ALIGNMENT&gt;</pre> <p>cartography constants.</p>
CONFIG_DATA	<pre>&lt;PREFIX&gt;_START_SEC_CONFIG_DATA &lt;ALIGNMENT&gt;</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CONFIG_DATA &lt;ALIGNMENT&gt;</pre> <p>configuration constants.</p>
CODE	<pre>&lt;PREFIX&gt;_START_SEC_CODE</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CODE</pre> <p>mapping code to application block, boot block, external flash etc.</p>
CALLOUT_CODE	<pre>&lt;PREFIX&gt;_START_SEC_CALLOUT_CODE</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CALLOUT_CODE</pre> <p>mapping callouts of the BSW Modules</p>
CODE_FAST	<pre>&lt;PREFIX&gt;_START_SEC_CODE_FAST[_&lt;NUM&gt;]</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CODE_FAST[_&lt;NUM&gt;]</pre> <p>code that shall go into fast code memory segments. The optional suffix [_&lt;NUM&gt;] can qualify the expected access commonness, e.g. typical period of code execution.</p>
CODE_SLOW	<pre>&lt;PREFIX&gt;_START_SEC_CODE_SLOW</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CODE_SLOW</pre> <p>code that shall go into slow code memory segments.</p>
CODE_LIB	<pre>&lt;PREFIX&gt;_START_SEC_CODE_LIB</pre> <pre>&lt;PREFIX&gt;_STOP_SEC_CODE_LIB</pre> <p>For code of library segments for BSW module or Software Component.</p>

Table 213 Memory Allocation Keywords