OSEK/VDX

OSEK Run Time Interface (ORTI)

Part A: Language Specification

Version 2.1

16. July 2001

This document is an official release and replaces all previously distributed documents. The OSEK group retains the right to make changes to this document without notice and does not accept any liability for errors.
All rights reserved. No part of this document may be reproduced, in any form or by any means, without permission in writing from the OSEK/VDX steering committee.

Preface

OSEK/VDX is a joint project of the automotive industry. It aims at an industry standard for an open-ended architecture for distributed control units in vehicles.

For detailed information about OSEK project goals and partners, please refer to the “OSEK Binding Specification”.

General conventions, explanations of terms and abbreviations have been compiled in the additional inter-project "OSEK Overall Glossary".

Regarding implementation and system generation aspects please refer to the "OSEK Implementation Language" (OIL) specification.

Table of Contents

1 Introduction 4
1.1 General Remarks 4
1.2 Motivation 4
1.3 Benefits 4
1.4 Organization of ORTI Specification 5
1.5 Language Definition 5
1.5.1 Scope 5
1.5.1.1 Representation 5
1.5.1.2 Tracing Dynamic Data 5
1.6 Acronyms 5
2 KOIL 6
2.1 File Structure 6
2.1.1 Version Section 6
2.1.2 Declaration Section 6
2.1.3 Information Section 6
2.2 Grammar 7
2.3 Version Section 8
2.4 Declaration Section 9
2.5 Information Section 9
2.6 Rules 10
3 Attributes 11
3.1 Attribute types 11
3.1.1 CTYPE 11
3.1.2 STRING 11
3.1.3 ENUM 11
3.1.3.1 Links 12
3.2 Attribute Description 13
3.3 Attribute Modifier 13
3.3.1 TOTRACE 13
4 History 14

1 Introduction

1.1 General Remarks

1.2 Motivation

In order to have the OSEK specification accepted by the customer it is important to have good application development support. This leads to a shorter design cycle and quicker time to market. In addition to good support from code generators, CASE tools and compilers it is also necessary to have sophisticated debuggers available that are capable of displaying and debugging the configuration of the OSEK components. For this purpose internal data of the OSEK component has to be made available to the tool.

Since OSEK is a standard that is being supported by many different suppliers, there is a need for a standard OSEK Run Time Interface (ORTI) to third party vendors that is valid for all OSEK suppliers. This will decrease the amount of work necessary for widespread support of the OSEK standard. Once a tool is “OSEK aware“ for one OSEK implementation it is “OSEK aware“ for all implementations.

1.3 Benefits

The OSEK Run Time Interface (ORTI) offers the following benefits to the Tool Vendors:

- Internal operating system data is visible to the tool
- Information on Task entry and exit available (if possible)
- Information on all important properties of the OSEK objects
- One common interface for any Microcontroller Platform and any OSEK Vendor
- ASCII interface for the ORTI file makes extensions easy and manageable

ORTI offers the following benefits to the OSEK vendors.

- Support by a great variety of development tools
- One common interface to debugging tools
- Better acceptance by the customer

ORTI offers the following benefits to the OSEK customer.

- Internal operating system data is visible
- Information on all important properties of the OSEK objects
- Enhanced debug tools
- Free choice of tools
- Free choice of OSEK products

1.4 Organization of ORTI Specification

The ORTI specification consists of parts A and B. Part A introduces the Kernel Object Interface Language (KOIL) used by ORTI and focuses on the grammar (syntax). Part B describes the OSEK specific (standard) objects and their semantics.

1.5 Language Definition

1.5.1 Scope

The OSEK Run Time Interface (ORTI) is intended as a universal interface for development tools to the OSEK Operating System. However, ORTI is also suitable for other static kernel environments. Therefore the specification has been divided into two parts. This part (A) describes the language used to define kernel objects to a debug tool. This language is called Kernel Object Interface Language (KOIL).

Part B describes OSEK-specific kernel objects, attributes and their semantics. So, ORTI uses KOIL as a vehicle to pass information about kernel objects to the debugger and specifies a number of semantic rules for standard objects, which are OSEK specific.

ORTI does not describe how the obtained data is to be represented on the screen - this is the responsibility of the debugger.

In order to keep ORTI universal and manageable it does not contain vendor specific functions. Information is provided via an ASCII text file. Since OSEK implementations are configured statically, the data is available at build time.

A dynamic representation of configuration data is not necessary with OSEK and therefore not part of ORTI.

1.5.1.1 Representation

Two types of data shall be made available to the debug tool. One type describes static configuration data that will remain unchanged during program execution. The second type of data shall be dynamic and this data requires to be re-evaluated each time the debugging tool wishes to display the information. The static information is useful for display of general information and in combination with the dynamic data. The dynamic data gives information about the current status of the system (e.g. when halted).

1.5.1.2 Tracing Dynamic Data

A debugger can decide to keep track of some (e.g. important) dynamic data variables. For example, it can be interesting to display the history of the "running task". Dynamic data can be accessed by the debug tool using two different methods – by means of run time data tracing (typically with trace-memory) or by data access on breakpoints.

1.6 Acronyms

EBNF Extended Backus-Naur Form

KOIL Kernel Object Interface Language

ORTI OSEK Run-Time Interface

OS Operating System

2 KOIL

2.1 File Structure

An ORTI file consists of the following parts: version section, declaration section and information section.

2.1.1 Version Section

This section describes the version of KOIL as well as the Kernel used. In case of ORTI, the Kernel version reflects the version number of the ORTI standard used.

2.1.2 Declaration Section

This section declares the kernel types (similar to a structure declaration in the C language) present in the implementation. It describes the method that should be used to access and interpret the data obtained for a kernel object of such a type. This section may also detail the suggested display name for a given attribute.

2.1.3 Information Section

This section contains information on all kernel objects (using types from the declaration section) that are currently available for a given system. It describes the method that shall be used to reference or calculate each required attribute. This information is either supplied as static values or else a formula that shall be used to calculate the required value.

2.2 Grammar

This section describes the grammar of the Kernel Object Interface Language (KOIL) in LL(1) using the Extended Backus-Naur Form. The terminal symbols marked with [1] are described in the paragraph Rules in this chapter

file	=	version_section declaration_section information_section ;
version_section	=	'VERSION' '{' koil_version kernel_version '}' ';' ;
koil_version	=	'KOIL' '=' ' "2.1" ' ';' ;
kernel_version	=	'OSSEMANTICS' '=' semantics_name ',' semantics_version ';' ;
declaration_section	=	'IMPLEMENTATION' implementation_name '{' { declaration_spec } '}' ';' ;
declaration_spec	=	object_type '{' { attribute_decl } '}' [',' type_description] ';' ;
attribute_decl	=	['TOTRACE'] attribute_type attribute_name [',' attribute_description] ';' ;
attribute_type	=	c_type | enum_type | string_type ;
c_type	=	'CTYPE' [ctype_decl] ;
enum_type	=	'ENUM' [ctype_decl] '[' enum_element { ',' enum_element } ']' ;
string_type	=	'STRING';
enum_element	=	enum_desc '=' (constant | formula) ;
information_section	=	{ object_def } ;
object_def	=	object_type object_name '{' attribute_def '}' ';' ;
attribute_def	=	attribute_name '=' formula ';' ;
implementation_name	=	koil_identifier[1] ;
object_name	=	koil_identifier[1] ;
object_type	=	koil_identifier[1] ;
attribute_name	=	koil_identifier[1] ;
link_name	=	koil_identifier[1] ;
semantics_name	=	string[1] ;
semantics_version	=	string[1] ;
ctype_decl	=	string[1] ;
enum_desc	=	string[1] [':' link_name] ;
attribute_description	=	string[1] ;
type_description	=	string[1] ;
constant	=	(integer_constant[1] | character_constant[1]) ;
formula	=	' " ' expression ' " ' ;
expression	=	logical_OR_expression { '?' expression ':' expression } ;

logical_OR_expression = logical_AND_expression { '||' logical_AND_expression } ;

logical_AND_expression = inclusive_OR_expression { '&&' inclusive_OR_expression } ;

inclusive_OR_expression = exclusive_OR_expression { '|' exclusive_OR_expression } ;

exclusive_OR_expression = AND_expression { '^' AND_expression } ;

AND_expression = equality_expression { '&' equality_expression } ;

equality_expression = relational_expression { ('==' | '!=') relational_expression } ;

relational_expression = shift_expression { ('<' | '>' | '<=' | '>=') shift_expression } ;

shift_expression = additive_expression { ('<<' | '>>') additive_expression } ;

additive_expression = multiplicative_expression { ('+' | '-') multiplicative_expression } ;

multiplicative_expression = cast_expression { ('*' | '/' | '%') cast_expression } ;

cast_expression = { '(' type_name ')' } unary_expression ;

unary_expression = postfix_expression
| unary_operator cast_expression
| 'sizeof' unary_expression
| 'sizeof' '(' type_name ')' ;

unary_operator = '&' | '*' | '+' | '-' | '~' | '!' ;

postfix_expression = primary_expression { '[' expression ']' | ('.' | '->') appl_identifier[1] } ;

primary_expression = appl_identifier[1] | constant | '(' expression ') ' ;

constant = integer_constant[1]
| character_constant[1]
| floating_constant[1]
| enumeration_constant[1] ;

type_name = { type_specifier } ['*'] ;

type_specifier = 'void' | 'char' | 'short' | 'int' | 'long' | 'float' | 'double'
| 'signed' | 'unsigned' | type_def_name ;

type_def_name = appl_identifier[1] ;

2.3 Version Section

In an ORTI file, the KOIL statement refers to the version used of Part A. The OSSEMANTICS statement contains a semantics name and the SEMANTICS version (e.g. "MYRTOS","1.0"). For ORTI-compliant OSEK kernels the semantics name must be “ORTI” and the version refers to Part B of the ORTI specification used. For example:

```
VERSION
{
        KOIL = "2.1";                   // KOIL (Part A), Spec v2.1
        OSSEMANTICS = "ORTI", "2.1";    // ORTI, Part B Spec v2.1
};
```

2.4 Declaration Section

The declaration section contains all (one or more) object type declarations of a certain kernel implementation:

```
IMPLEMENTATION MYOSEK
{
        <object type declarations>
};
```

Each object type declaration contains the type of its attributes (similar like a C structure declaration contains the type of its members). The complete set of attributes must be declared in one single object type declaration. The attribute type informs the debugger how to access the target memory and interpret the value for display. For example (declaration of the object 'TASK' having two ENUM type attributes):

```
TASK
{
        ENUM
        [
                "RES_SCHEDULER" = 0,
                "8" = 1,
                "4" = 2
        ] PRIORITY, "Actual Prio";
        ENUM
        [
                "READY"=0, "RUNNING"=1, "WAITING"=2, "READY"=3, "SUSPENDED"=4
        ] STATE, "State";
};
```

2.5 Information Section

The information section consists of all (one or more) object definitions ('instantiations'). Every object definition refers to an object type from the declaration section. The complete set of attributes must be defined in one single object definition. The attribute definition contains the formula (a debugger expression, see2.2) required to retrieve the value. No attribute definition may appear that is not declared in the object type declaration. For example (defining two 'TASK' objects):

```
TASK SampleTaskFirst
{
        PRIORITY = "osTcbActualPrio[0]";
        STATE = "osTcbTaskState[0]";
};

TASK SampleTaskSecond
{
        PRIORITY = "osTcbActualPrio[1]";
        STATE = "osTcbTaskState[1]";
};
```

2.6 Rules

An ORTI file must conform to the following:

- All objects are described using the KOIL syntax.
- Each object-type and object-name must have a unique name.
- An ORTI file may contain C++ -style comments (`/* */` and `//`), where C++ rules apply.
- Whitespace (blank, CR, LF, TAB, comment) between terminals is ignored.
- All keywords and identifiers are case-sensitive.
- The `integer_constant` terminal represents a number, where the standard C convention is used for decimal, hexadecimal and octal notation.
- The `character_constant` terminal follows the C definition as well, including the support of all standard escape sequences, such as '\n', '\t' etc.
- The `floating_constant` terminal follows the C definition.
- The `string` terminal (also known as *string constant* or *string literal*) represents a sequence of zero or more characters surrounded by double-quotes ("). The C string definitions apply (not including escape sequences except \" and \\). String constants can be concatenated, as in: "aap" "noot", being equivalent to "aapnoot".
- The `appl_identifier` terminal represents any ISO/ANSI-C identifier and represents application symbols. These symbols rely on symbolic information retrieved from the debug information of the application and must have 'external linkage' scope (e.g. global C variables). The symbol value is only valid after the application has executed its initialization phase (typically this is the system startup code before reaching the applications entry point, which is main() in C). The only exception to this constraint is when using the unary address-operator (&).
- The `koil_identifier` terminal represents KOIL names. The syntax of <koil_identifier> is the same as the syntax of the ISO/ANSI-C identifier.
- A formula may contain the address of the variable or some expression that is interpreted by the debugger to evaluate the attribute value. The formula is represented as a subset of C-expressions.
- All formulas in the declaration section must yield the correct value immediately after download of the application, and must yield the same value whenever calculated after download.
- All formulas in the information section may be dynamic; tools are free to evaluate these values as appropriate.
- The debugger should have enough information available to be able to resolve the C expression.

3 Attributes

3.1 Attribute types

Any attribute type has to be declared inside an object declaration, before it may be used (referred to) inside an object definition. An attribute type uses one of the following KOIL data types.

3.1.1 CTYPE

CTYPE is a generic type that corresponds to the High-Level Language (HLL) debug information type of the expression that defines the value of the attribute.

```
CTYPE PRIORITY;
```

Since some ORTI consuming tools might not have access to the full debugging information of the application, optionally a tentative type for the attribute can be specified, as in:

```
CTYPE "unsigned char" PRIORITY;
```

This tentative type must be a valid C-type within the application, for non-ANSI type extensions the type information must be contained in the debug information. This type could be used as a type cast by the ORTI consuming tool, when evaluating a formula. When both the HLL debug type information as well as the tentative type are present, the tentative type prevails.

3.1.2 STRING

The value of the STRING attribute should be displayed by the debugger as supplied. The rule of <string> applies. For example:

```
STRING HOME_PRIORITY, "Home Priority";
```

3.1.3 ENUM

ENUM defines a table matching internal attribute values to a display string that should be used by the debugger to display a state to the user. An ENUM table describes the value interpretation for one particular attribute. The exact type of the internal attribute value shall be defined after the 'ENUM' keyword.

```
ENUM ["RUNNING" = 1, "READY" = 2, "SUSPENDED" = 0, "WAITING" = 3] STATE;
```

This means that if the formula for the state attribute evaluates to the value "3" then the descriptive text "WAITING" should be displayed rather than the numeric value 3.

```
ENUM ["10" = 1, "20" = 2, "25" = 3] PRIORITY;
```

This means that if the formula for the priority attribute evaluates to the value "2" then the value "20" would be displayed.

```
ENUM ["TaskA" = "&(taskCB[3])", "TaskB" = "&(taskCB[4])"] RUNNINGTASK;
```

This example shows that formulas could also be used in ENUM definitions. As stated in section 2.6 the formula in the declaration section will be evaluated only once. Therefore take care about using variable contents in the declaration section like in the following example:

```
ENUM ["TaskA" = "taskCB[3].id", "TaskB" = "taskCB[4].id"] RUNNINGTASK;
/* Not recommended! */
```

Since some ORTI consuming tools might not have access to the full debugging information of the application, optionally the tentative type of the attribute can be specified, as in:

```
ENUM "unsigned char" ["10" = 1, "20" = 2, "25" = 3] PRIORITY;
```

This tentative type must be a valid C-type within the application, which could be used as a type cast by the ORTI consuming tool, when evaluating a formula. When both the HLL debug type information as well as the tentative type are present, the tentative type prevails.

If the value does not belong to the enumeration the value with some error indication should be displayed.

3.1.3.1 Links

Links are an optional extension of ENUM and represent a reference to another kernel object whose definition is in the same file.

```
IMPLEMENTATION myOSEK
{...
    TASK
    {  ...
       ENUM
       ["First Stack" : Stack1 = "&taskStack1[0]",
        "Second Stack" : Stack2 = "&taskStack2[0]"
       ] STACK;
        ...
    };

    STACK
    {
       ...
    };
};


// Definitions of the STACK objects which may be referenced by TASK objects

TASK TaskA
{
    STACK = "taskA.stack";
};


STACK Stack1
{
    ...
);


STACK Stack2
{
    ...
);
```

3.2 Attribute Description

Attributes can have an additional description. These descriptions are optional and may be added at the end of every object attribute declaration. Description fields start after a comma (,) and use the <string> terminal. For example:

```
CTYPE PRIORITY, "Priority";
```

Descriptions allow the possibility of supplying more relevant names to a debugger without limiting the possible names that an attribute may have. The descriptions supplied are typically implementation specific.

3.3 Attribute Modifier

3.3.1 TOTRACE

The attribute modifier TOTRACE informs the debugger that the attribute should be traced by the debugging tool, if this is feasible.

4 History

Version	Date	Remarks
1.0 – 2.0		Not published by OSEK/VDX
2.1	16. July 2001	Authors: Mr. Barthelmann (3SOFT) Mr. Büchner (Hitex) Mr. Dienstbeck (Lauterbach) Mr. Elies (Hitex) Mr. Fathi (Cosmic) Mr. Hoogenboom (Green Hills) Mr. Janz (Vector) Mr. Kriesten IIIT, (University of Karlsruhe) Mrs. Nieser (Lauterbach) Mr. Nishikawa (Toyota, Europe) Mr. Schimpf (ETAS) Mr. Stehle (Vector) Mr. Ulcakar (iSystem) Mr. Vetterli (Metrowerks) Mr. Wertenauer (Cosmic) Mr. Winters (Motorola)