EDIABAS

Electronic Diagnostic Basic System

BEST/2 LANGUAGE DESCRIPTION

VERSION 6b

CONTENTS

1. Introduction to BEST/2

BEST is the acronym for '**BE**schreibungssprache für **ST**euergeräte' (description language for control units). In EDIABAS this description language is used to meet the requirement for an application-independent description facility. During the program runtime the system loads and interprets description files in which information about the control units such as addresses and conversions is hidden (encapsuling). BEST/2 description language therefore provides the means to convert data stored in the control units into control unit independent values such as engine speed or control unit number. Only data actually required by the application is downloaded (information hiding).

A language with a syntax based on assembler language, BEST/1, already exists for this purpose. However although BEST/1 is good for test purposes and simple programming it is difficult to read and understand.

The more readable BEST/2 language was developed based on C language to resolve these problems. C was chosen because it is in widespread use, is structured and contains only a few language elements that are easy to learn.

BEST/2 is a problem-oriented language, with special functions that are called by library functions. The only exception to this rule is the result manager which is a fundamental part of the language definition.

There are no storage definitions because the virtual processor used with BEST/2 has no memory apart from the stack that is already available in the processor.

2. Text conventions

There are seven classes of word: names, reserved words, constants, strings, lists, operator and other separators. Blanks, tabs, line separators and comments are classed as spaces and ignored apart from the fact that they are needed to separate adjacent names, constants and reserved words.

2.1. Comments

A comment starts with the characters /* and ends with the characters */. Comments that start with the characters // end at the end of the line. Comments by /* */ cannot be nested.

2.2. Names

A name consists of a sequence of letters and digits; the first character must be a letter. The underscore character _ counts as a letter. There is case sensitivity. The first 32 characters are used to distinguish names, but they may also be longer.

2.3. Reserved words

The following words are reserved and can only be used with their predefined meaning:

argument	**author**	**break**	**case**	**char**
comment	**const**	**continue**	**data**	**default**
defrslt	**do**	**ecu**	**else**	**exist**
if	**int**	**job**	**language**	**long**
name	**origin**	**range**	**real**	**result**
return	**revision**	**size of**	**string**	**switch**
type	**unsigned**	**uses**	**while**	**##asm**
#define	**#endasm**	**#include**	**#undef**	

2.4. Constants

There is a whole range of constants and they are described in this section. Section 2.5 describes the properties of the virtual processor that affects the size of the constants. All constants are Long Constants.

2.4.1. Integer constants

An integer constant consists of a string of digits. Its type is **int** and it is usually interpreted decimally.

If the string starts with the sequence **0x** or **0X** it is interpreted hexadecimally, i.e. in base 16, and the letters **a** (or **A**) to **f** (or **F**) count as hexadecimal digits with the decimal values 10 to 15.

If the string starts with the sequence **0y** or **0Y** it is interpreted binary, i.e. in base 2, and only digits **1** and **0** are used.

2.4.2. Char constants

A character in single quotes, e.g. **'x'**, is a **char** constant and its value is the value of the character in the ASCII character set.

2.4.3. String constants

A string is a sequence of characters in double quotes, i.e. **"..."**. A string ends with the character **0**.

2.4.4. Lists (Data) constants

A list is a sequence of other constants (not lists) separated by **,** and enclosed in **{** and **}**. Strings in lists do not end with **0**. The length of the list carried over.

2.5. Sizes

char	8 bit ASCII
int	16 bit
long	32 bit
real	64 bit
char[]	max. 1023 bytes
int[]	max. 1023 bytes
long[]	max. 1023 bytes

2.6. Syntax notation

The following rules apply when describing the grammar: "I" separates alternatives, "[" and "]" enclose optional parts, "{" and "}" enclose optional parts that can be specified more than once, and ";" terminates a rule.

2.7. What does a name mean

A name is an object known to the compiler by its type and memory location.

2.8. Objects and L values

An object is a storage area that can be modified. An L value is an expression that describes an object. A name is a simple example of an L value expression. The term L value recalls the assignment **E1=E2** in which the left operand must be an L value expression.

3. Conversions

3.1. Characters and integers

char, **int** and **long** can be used wherever an integer object is needed. The values are converted to integer values. A **char**, **int** or **long** value has a sign that is retained when converted to longer integer values.

3.2. Unsigned integers

When an **unsigned** value and a regular value are combined the regular value is converted to an **unsigned** value and the result is also **unsigned**. A short **unsigned** value is converted into a regular value by adding zero bits as the most significant bits.

3.3. Arithmetic conversions

First **char** values are changed into **long** values.

If one of the operands is **unsigned** then the other is also changed to **unsigned**, and this then also becomes the type of the result.

Otherwise, both operands are **long** values and the result is also a **long** value.

4. Expressions

This section describes the operators for expressions. For simple expressions and unary operations the manual states whether they return L values. The other operators do not return L values.

The following sections are arranged in descending order of priority of the operators; all the operators in a section have equal priority. The associativity of the operators is given in each section (invisible bracketing).

Apart from priority, the processing sequence is *undefined*.

NOTE: Operators cannot be used with real values since this will lead to undefined results! Real values can only be processed with the corresponding BEST/2 functions.

4.1. Simple expressions

Simple expressions are references to objects and constants, the index operation and function calls. These operations are left-associative.

```
primary:
          identifier
          | constant integer
          | constant array
          | (expression)
          | primary ([argument list])
          | primary [expression]
          ;

 argument list:
          assignment {, assignment}
          ;
```

A name is a simple expression provided it has been suitably declared. Most names are L values. The name type is usually given by the declaration. A constant is an L value. Constant arrays always return 1. Identifiers return their value. Arrays always returns the value 1 unless they are assigned to arrays. Names of job results return 1 if the result was requested, otherwise 0. Names of job parameters return 1 if they were specified, otherwise 0.

4.2. Unary operators

Unary operators are right associative.

```
 unary:
```

```
primary
| primary++
| primary--
| - primary
| ! primary
| ~ primary
| ++ primary
| -- primary
| exist identifier
| sizeof unary
| sizeof (type specification)
```

The unary operator produces the negative value of its operand.

The ! operator for logic negation returns 1 for an operand with the value 0 and 0 for all other operands. The result is the long type.

The ~ operator complements the individual bits in its operand.

The ++ and -- operands (sic) modify their operands by adding or subtracting 1. The operand must be an L value. The result is not an L value.

The expression ++E is equivalent to E+=1, so the result is the new value of the operand after 1 is added.

The expression --E is equivalent to E-=1, so the result is the new value of the operand after 1 is subtracted.

The expression E++ returns the same value of operand E, i.e. the original value of the object that the L value E describes. 1 is added to the object itself.

The expression E-- returns the same value of operand E, i.e. the original value of the object that the L value E describes. 1 is subtracted from the object itself.

The sizeof operator returns the size of its operand measured in bytes.

The exist operator returns <>0 (TRUE) or 0 (FALSE) depending on whether the specified identifier is available or not. This is always TRUE for all variables, constants and result names but only for job arguments when they were defined when the job was called.

4.3. Multiplication and division

The operators ***,/** and **%** for multiplication and division are left associative. The usual arithmetic conversions are used.

```
multiplication:
     unary
     | multiplication * unary
     | multiplication / unary
     | multiplication % unary
     ;
```

The binary operator ***** designates multiplication. It is commutative and associative.

The binary operator **/** designates division. When positive values are divided it breaks off towards 0. Otherwise it breaks off towards the most negative number.

The binary operator **%** returns the remainder after dividing its two operators.

4.4. Addition and subtraction

The operators **+** and **-** for multiplication and division are left associative. The usual arithmetic conversions are used.

```
addition:
        multiplication
        | addition + multiplication
        | addition - multiplication
        ;
```

The + operator returns the sum of its operands. It is commutative and associative.

The - operator returns the difference of its operands.

4.5. Shift operations

The shift operators << and >> are left associative. The usual arithmetic conversions are used. The right operand is converted into an **unsigned** value.

```
shift:
        addition
        | shift << addition
        | shift >> addition
        ;
```

The value of **E1<<E2** is the bit pattern of **E1** shifted **E2** bits to the left.

The value of **E1>>E2** is the bit pattern of **E1** shifted **E2** bits to the right. If **E1 unsigned** then bits with the value 0 are inserted. If **E1** is a regular value then bits are inserted according to its bit sign.

4.6. Comparisons

Comparisons are left associative.

```
comparison:
            shift
            | comparison < shift
            | comparison <= shift
            | comparison > shift
            | comparison >= shift
            ;
```

The comparison operators return the **long** value 0 if the specified relation is false and 1 when the relation exists. The usual arithmetic conversions are used.

4.7. Equality comparisons

The operators **==** and **!=** behave in the same way as the other comparison operators but have a lower priority.

```
equality:
            comparison
            | equality == comparison
            | equality != comparison
            ;
```

4.8. AND gating of bits

```
bit-and:
            equality {& equality}
            ;
```

The **&** operator is commutative and associative. The usual arithmetic conversions are used.

4.9. Exclusive OR gating of bits

```
bit-exclusive-or:
            bit-and {^ bit-and}
            ;
```

The **^** operator is commutative and associative. The usual arithmetic conversions are used.

4.10. OR gating of bits

```
bit-or:
            bit-exclusive-or {³ bit-exclusive-or}
            ;
```

The | operator is commutative and associative. The usual arithmetic conversions are used.

4.11. Logic AND gating

andif:
  bit-or {**&&** bit-or}
  ;

The **&&** operator is left associative. The result is 1 when both operators are not 0, otherwise the result is 0. The right operand is only evaluated when the left is not 0.

4.12. Logic OR gating

binary:
  andif {³³ andif}
  ;

The || operator is left associative. The result is 0 when both operators are 0, otherwise the result is 1. The right operand is only evaluated when the left is 0.

4.13. Assignments

Assignment operators are right associative. The left operand must always be an L value. The type of the result is always the type of the left operand. An assignment operation returns the value in the left operand as the result.

assignment:
  binary
  | unary **=** assignment
  | unary ***=** assignment
  | unary **/=** assignment
  | unary **%=** assignment
  | unary **+=** assignment
  | unary **-=** assignment
  | unary **&=** assignment
  | unary **^=** assignment
  | unary **|=** assignment
  ;

With simple assignment the value of the right operand replaces the value of the object which designates the left operand. Before assignment the right operand is converted to the type of the left operand.

The value of the assignment in the form E1 op= E2 can be derived from the assignment **E1 = E1 op (E2)**. **E1** is only evaluated once however.

4.14. Lists of expressions

```
expression:
        assignment {, assignment}
        ;
```

Two expressions separated by a comma are evaluated left to right. The type and value of the result are the type and value of the right expression. This operation is left associative.

5. Declarations

Declarations define how BEST/2 interprets the individual names entered by the user. They take the following form:

declaration:
 type-name initialized-declarator-list;
 | **real** identifier;
 ;

Definitions reserve storage space (register) and contain information for the type of a list of declarators. The declarators contain the names that are declared, possibly together with initializers.

5.1. Type names

The following type names exist:

type-name:
 [unsigned]int
 | **[unsigned]long**
 | **[unsigned]char**
 ;

extra-type-name:
 data
 string
 real
 ;

A declaration can only contain one type name.

5.2. Declarators

A declaration contains a list of declarators separated by commas. In data definitions an initializer can also be specified after each declarator.

initialized-declarator-list:
 declarator [=initializer]
 ;

declarator:
 identifier {'**[**' '**]**'}
 | (declarator)
 ;

If a declarator is followed by a pair of square brackets then an array variable was declared.

5.3. Initializers

Data can also be initialized in a definition. Each initializer is preceded by the character = and followed by an expression.

```
initializer:
        assignment
```

5.4. Predefined Macros

BEST/2 contains a number of predefined macros permitting the query of header definitions in BEST/2 jobs:

__ECU__	designation of the control unit
__ORIGIN__	author of the first version
__REVISION__	current version (corresponds to header definition revision)
__AUTHOR__	author
__LANGUAGE__	language
__USES__	basic description files
__ECUCOMMENT__	comment (1st line)

In BEST/2 jobs, the macros are to be used like global string constants. If one of the optional header definitions is not available, the macro will supply a null string.

6. Statements

Unless otherwise specifically stated, statements are executed sequentially.

statement:
statement-prefix statement
| [expression];
| compound-statement
| **if** (expression) statement
| **if** (expression) statement **else** statement
| **do** statement **while** (expression);
| **break;**
| **continue;**
| **return;**
;

statement-prefix:
while (expression**)**
| **switch (**expression**)**
| **case** constant-integer;
| **default:**
;

6.1. Compiler instructions

All compiler instructions must be indicated at the beginning of a line.

6.1.1. #include

With the **#include** instruction, additional BEST/2 source files can be entered in BEST/2 description files. A line

#include "BEST/2 source file"

is replaced by the content of the desired file. If absolute and relative paths are indicated, DOS notation must be used.

If the indicated Include file name contains an absolute or relative path, the Include file must be located in the corresponding directory. With relative paths, the path of the BEST/2 source file is used.

If the indicated Include file name contains no path, searching starts in the directory of the BEST/2 description file. With the compiler, the search can be extended to other directories.

Behind the filename of an #include instruction only a comment is permitted. The comment has to end within the same line. A comment over more than this line is not permitted.

6.1.2. #define

In BEST/2 description files, text replacements can be defined with the instruction **#define**. A line

#define identifier text

causes the name to be replaced by the indicated text throughout the BEST/2 description file. It is possible to distribute a long replacement text over several lines by placing the character \ at the end of the line. No text will be replaced within comments and character strings.

The name is structured according to the rules for other BEST2 names.

Macros and Escape symbols are not supported.

Behind the text of an #define instruction only a comment is permitted. The comment has to end within the same line. A comment over more than this line is not permitted.

6.1.3. #undef

In BEST/2 description files, previously performed #define definitions can be undone with the instruction

#undef identifier.

Behind the identifier of an #undef instruction only a comment is permitted. The comment has to end within the same line. A comment over more than this line is not permitted.

6.1.4. #asm #endasm

In BEST/2 description files, BEST/1 code can be entered. It will be inserted between the instructions **#asm** and **#endasm**.

```
#asm
     BEST/1 code
```

#endasm

6.2. Assignments and procedure calls

The calculation of an expression is the most commonly used statement; it takes the following form:

expression;

Such statements are usually assignments or procedure calls.

6.3. Blocks

A number of statements are assembled in a block. A block can always be specified instead of a single statement.

```
compound-statement:
        {{declaration}{statement}}
        ;
```

A block can contain declarations. If a name that was previously entered is declared then the previous declaration is replaced for that area of the block and restored at the end of the block. This does not apply to array variables which apply globally for all blocks as soon as they are defined.

6.4. "if" statements

There are two types of **if** statement:

```
        if(expression)
        statement
```

and

```
        if(expression)
        statement
        else
        statement
```

In both cases the expression is evaluated first. If the result is not 0, then the first dependent statement is executed. In the second case the second dependent statement is executed when and only when the result is 0. **Else** is assigned to the next **if**.

6.5. "while" statement

The **while** statement has the following form:

while(expression)
statement

The dependent statement is repeated for as long as the value of the expression is not 0. The expression is evaluated each time before the dependent statement is executed.

6.6. "do" statement

The **do** statement has the following form:

do
statement
while(expression);

The dependent statement is repeated for as long as the value of the expression is not 0. The expression is evaluated each time after the dependent statement is executed. That means, the statement is executed at least once.

6.7. "switch" statement

The **switch** statement ensures that the execution of the program is continued with one or more statements depending on the value of an expression. The **switch** statement has the following form:

switch(expression)

statement;

The usual arithmetic conversions are used to evaluate the expression but the result must be an integer. The dependent statement is typically a block. Each statement within the dependent statement can be preceded by any number of **case** labels:

case constant-integer:

Each of these constants can only occur once in a **switch** statement. A **case** label can also take the following form:

default:

To execute the **switch** statement the expression is evaluated and compared with all **case** constants. If a **case** constant is found that has the same value as the expression, then the execution of the program is continued with the statement that follows the **case** label. If no suitable **case** constant is found then execution is continued at a **default case** label if one exists.

6.8. "break" statement

The

break;

statement aborts the nearest **do-**, **while-** or **switch** statement.

6.9. "continue" statement

The

continue;

statement must be dependent on a **do-** or **while-** statement. It ensures that the execution of a program is continued from the point where the repeat of the nearest statement is decided.

6.10. "return" statement

If a job was called, the **return** statement within the job assures that exit is made from the job again. A string variable or string constant can be specified along in the **return** statement. The runtime system assigns this value to the result JOBSTATUS in result record 0. The **return** statement syntax is shown below:

return {constant-string **|** string-expression};

7. Global declarations

A BEST/2 description file consists of a header and a sequence of jobs:

```
program:

        header
        { declaration }
         job
```

7.1. Header

The definition of a header is structured as follows:

header:

ecu	**:**	string;
origin	**:**	string;
revision	**:**	string;
author	**:**	string;
language	**:**	string;
uses	**:**	string;
comment	**:**	string;]

ecu specifies the exact designation of the ECU.

origin specifies the author and initial creation..

revision specifies the version as numeric pair separated by a dot.(e.g. 1.0). Both numbers has to be within the range 0-65535.

author specifies the author of the last change.

language specifies the language (optional).

uses specifies the basic description files (optional).

comment specifies a user comment, whereby several comment lines may be specified.

The header definitions can be queried in the BEST/2 jobs of the corresponding description file by means of PREDEFINED MACROS.

7.2. Jobs

The definition of a job takes the following form:

job:

```
        job-header compound-statement

        ;
```

job-header:

```
        job     ( name        :     identifier;
                { comment     :     string;}
                {[ argument   :     identifier;
                   type       :     type-name|extra-type-name;
                { comment     :     string;}]}
                {[ result     :     identifier;
                   type       :     type-name|extra-type-name;
                   range      :     value|string;
                   defrslt    :     value|string;
                { comment     :     string;}]}
                )
```

The job header defines the name of the job, the call parameters and possible results. The parameters are declarations with possible initializers. These initializers are then evaluated as default results or call parameters. The declaration of default results is optional.

8. Areas of application

Global declarations apply to all jobs within the description file. **These global declarations declare constants!** These cannot be modified!

Otherwise declarations only apply to the block in which they are described.

9. Runtime library

The runtime library provides functions for communication, string handling, error handling etc. The runtime library functions are written in BEST/1 so cannot be created by the user himself.

They are described in the "BEST/2 FUNCTION PRIMER" manual.