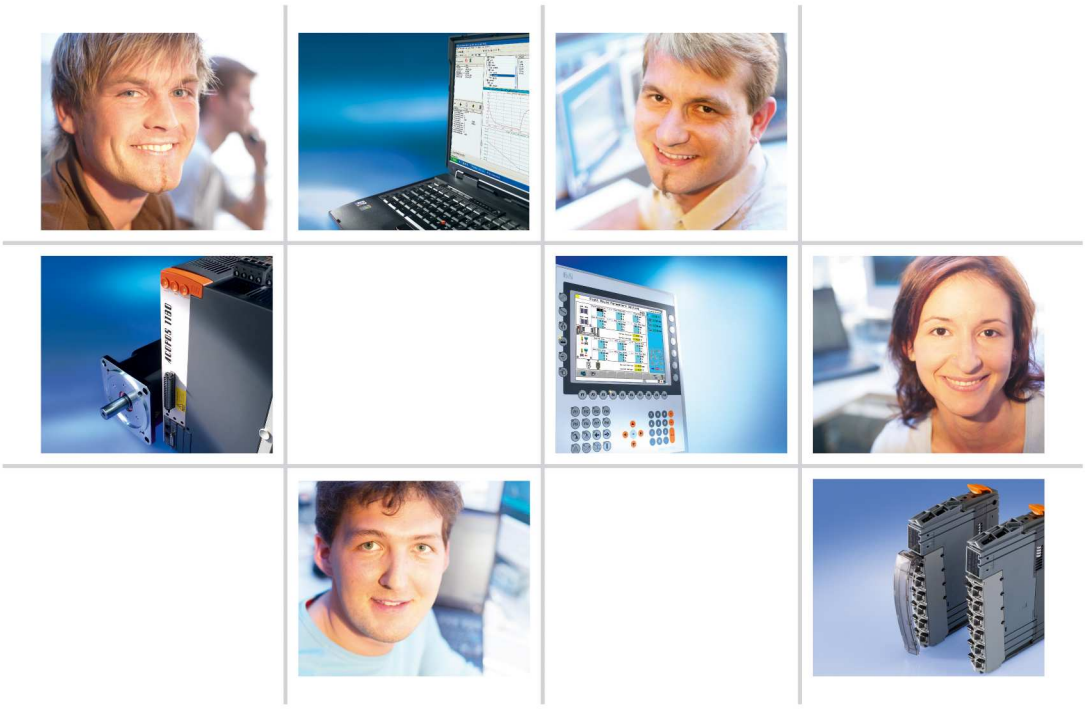
B&R Coding Guideline



Perfection in Automation

www.br-automation.com

Prerequisites

Software:	Automation Studio
Hardware:	none

Table of Contents

1. INTRODUCTION 4

2. B&R CODING GUIDELINES 5

2.1 Before You Code 6
2.2 Naming Conventions 6
2.3 Code Format 10
2.4 Programming Techniques 13
2.5 Testing 15
2.6 Documentation 16

1. INTRODUCTION

This document is about generating application software in the field of automation. If you are (or are going to be) a programmer of machines or plants, please ask yourself a few questions:

- Is software generation more than just coding, coding, coding?
- How can I improve the quality of the software I produce?
- By the way, what is software quality?
- What about costs to fix defects in software?
- How do I create well structured software?
- Is there a way to analyze, describe and discuss machine logic in a formal and exact way?
- How can I write better source code?
- How should I test and document the code I create?

The section “B&R Coding Guidelines” presented the B&R Coding Guidelines for automation application software which should guide the user in developing a programming style to produce, test and document high quality source code.

2. B&R CODING GUIDELINES

Computer programming is an engineering discipline (software engineering) and as usual in engineering there **is** an absolute truth ... whether a program **does** work or it **does not** work.

But computer programming also is an art (see the famous book 'The Art of Computer Programming' by Donald E. Knuth which has been named among the best twelve scientific monographs of the century) as sometimes it is more a question of aesthetics how a program does it's job and if the code looks appealing. Without question programming is a creative process.

Software production costs money - and - earns you money. It is the B&R philosophy to produce high quality products, and software is no exception here. So let's produce high quality software code!

Attributes of high quality code are (among others):

- clean architecture and design
- easy to read and understand
- easy to maintain
- re-usable
- well commented
- bug free

This document should assist you in improving your code quality. If you follow the guidelines outlined here your code should be of reasonable quality.

You are working in a team so please be considerate of your colleagues, who maybe won't appreciate dealing with those quick'n'dirty completely undocumented routines you hacked at 2:00 am Saturday night.

In the end **you** (the author) are responsible for the code you create. Do it well and then be proud of what you have created and achieved!

2.1 Before You Code

The foundations of any good software are a clean architecture and design. Take your time in the conception phase and work on the design until you are happy with your software concept.

Before you actually code please mind the concepts and methods discussed in sections of this document.

2.2 Naming Conventions

Good variable and data type names are a key element of program readability. All names should be descriptive and easy to read. Use either underscores or capital letters (don't mix them) in composite names to enhance readability, like

```
actPressure = actForce / pistonArea;
cmdCount++;
```

or

```
act_pressure = act_force / piston_area;
cmd_count++;
```

An identifier may contain letters and numbers and must start with a letter. You cannot use reserved keywords as identifiers. A complete list of reserved key words for each programming language can be found in the Automation Studio online help.

2.2.1 Language

If no different specification is given by the customer it is strongly recommended to code and comment in English for trouble-free international usage of software. Within B&R this recommendation is compulsory.

2.2.2 User Data Types (Structures)

User data types start with an upper case letter and end with the postfix '_type', in between lower and upper case letters may be mixed.

```
TYPE
    Recipe_type:  STRUCT
        base:     UINT;
        binder:   UINT;
        additive: USINT;
    END_STRUCT;
    MachineParams_type: STRUCT
        speed:       REAL;
        pressure:    REAL;
        temperature: INT;
        pRecipe:     REFERENCE TO Recipe_type;
    END_STRUCT;
END_TYPE
```

2.2.3 Constants

Constants are all upper case. Use underscores '_' to enhance readability.

```
VAR CONSTANT
    STEP_CONDITIONING: USINT := 23;       (* [-] *)
    HEATING_TIME_OUT:  UINT  := 5000;     (* [s] *)
    MAX_PRESSURE:      REAL  := 6.7e+006; (* [Pa]*)
END_VAR
```

These rules also apply to constants defined by the `#define` pre-processor directive and the enum statement in C source code. Please note that declarations via the `#define` directive and `enum` statement are local to the scope of your C code!

2.2.4 Local Variables

Local variables start with a lower case letter. Upper case-letters (or underscores '_') are only used to enhance readability.

```
VAR
    machineStep:    USINT; (* [-]     *)
    actPressure:    REAL;  (* [bar]   *)
    avgTemperature: INT;   (* [0.1°C] *)
END_VAR

VAR
    machine_step:    USINT; (* [-]     *)
    act_pressure:    REAL;  (* [bar]   *)
    avg_temperature: INT;   (* [0.1°C] *)
END_VAR
```

In the following, only examples without underscores are included. If you prefer naming with underscores you will be able to figure it out.

2.2.5 Global Variables

Global variables start with the pre-fix 'g' followed by an upper-case letter or '_':

```
VAR
    gHeaterOn: BOOL;
    gActCmd:   Cmd_typ;
    gCmdCount: UINT;
END_VAR
```

This convention is reserved for global variables - do not use it for non-global variables.

2.2.6 Pointers

Local pointers start with the pre-fix ‘p’ followed by an upper-case letter or ‘_’. Global pointers start with the pre-fix ‘gp’ followed by an upper-case letter or ‘_’:

```
VAR
    pActRecipe: REFERENCE TO Recipe_type;
END_VAR

VAR
    gpActRecipe: UDINT;
END_VAR
```

The above convention is reserved for pointers - do not use it for other variables.

Global pointers have the data type ‘UDINT’ because IEC doesn’t support pointers to generic data types. Cast the global pointer to a generic local pointer to access structure members:

```
pActRecipe = (Recipe_type*)gpActRecipe;
actSpeed   = pActRecipe->speed;
```

2.2.7 Hardware-Connected Variables

Variables assigned to hardware I/O points start with a pre-fix defining the I/O point type:

Prefix	Type
di	digital input
do	digital output
ai	analog input
ao	analog output

The pre-fix is followed by an upper-case letter or ‘_’:

```
VAR
    diEmergencyOff: BOOL;
    doSolidStateOn: BOOL;
    aiActTemp:      INT;  (* [0.1°C]                  *)
    aoValvePos:     INT;  (* 0=closed, 32767=open     *)
END_VAR
```

This convention is reserved for HW connected variables - do not use it for other variables.

2.2.8 C-local Variables

Variables defined in C source code are not visible outside their definition scope – you cannot see them e.g. in a Watch or Trace window.

```
Recipe_type* pPrevRecipe = 0;
USINT        cmdCount    = 0;
REAL         actSpeed    = 0; /* [m/s] */
```

If not declared ‘static’ they are allocated from stack each time the C routine is executed (and therefore non-remanent) and are not initialized! It is therefore wise to initialize them in the declaration (as done above).

2.2.9 Instances of Function Blocks

Instances of function blocks should be named to contain the name of the function block:

```
VAR
    valveSwitchTON: TON_type;
    solidStateTOF:  TOF_type;
    pressureLCPID:  LCPID_type;
END_VAR
```

2.3 Code Format

Visual layout of the code should accurately represent the logical structure of a computer program. Thus visual information acquisition of the human brain can support the reader in code understanding.

2.3.1 Indentation

Proper indentation is a key element for the readability of a code and is a **must** in all programs!

The whole idea behind indentation is to clearly visualize where a block of control starts and ends.

A large indentation size (6 or 8 characters) makes the code structure easier to see, while a smaller indentation size (2 or 4 characters) saves space on the right hand side of your screen.

We suggest an indentation size of 4 characters. If you have good reasons choose another indentation size that you prefer and stick to it.

2.3.2 File Header

Every file must have a header, which includes:

- Information about author and copyright
- Short description (summary comments) with a focus on purpose of the code, input and output variables, global effects of the routine, limitations and interface assumptions
- Timing behavior and memory requirements (if critical)
- Revision number, history and date (in an international unmistakable format, e.g. 04-March-2005 instead of 04-03-05 or 03/04/05)

A template header is automatically included when you create a program in Automation Studio™.

Revision number format is Vxx.yy, where xx is incremented with every major code update (e.g. when new features are added or incompatibilities to the previous version are introduced) and yy is incremented with minor improvements and bug fixes.

2.3.3 Placing Braces

There are a lot of brace placement strategies around. The preferred method is putting each brace on a line by itself combined with proper indentation:

```
if (inst.request > 0)
{
    inst.ok2jump = 1;
    inst.status  = 0;
}
else
{
    inst.ok2jump = 0;
    inst.status  = 5;
}

UINT CheckStatus(REAL xDeviation, REAL yDeviation)
{
    function body
}
```

If this doesn't look visually appealing to you, choose another consistent style, e.g. as suggested by Kernighan and Ritchie:

```
if (inst.request > 0) {
    inst.ok2jump = 1;
    inst.status  = 0;
}
else {
    inst.ok2jump = 0;
    inst.status  = 5;
}
```

2.3.4 Spaces

For readability reasons add a space before and after each operator:

```
xAxisPos = x0 + deltaX;
if (machineState == STATE_RUN)
   ...
```

with exception of:

.	member selection operator
->	member selection operator
[]	subscription operator
()	function call and function declaration operator
(type)	unary casting operator
++	pre- and post increment operator
--	pre- and post decrement operator
!	unary negation operator
~	unary one's complement

If the assignment operator '=' is placed directly behind the variable, a search (or search and replace) in the editor for e.g. 'someVariable=' will only find assignments to this variable in the code. If this is important for you, format assignments this way:

```
xAxisPos= x0 + deltaX;
```

We recommend placing the reference '&' and dereference operators '*' near the type in declarations:

```
void GetCtrlParams(REAL deadTime, REAL dXmax, Params_type*
pCtrlParams)
```

2.3.5 Visual Alignment

Visual alignment of elements that belong together reinforces the visual binding of these elements:

```
stPar.X0     = pIntern->X0;
stPar.deltaX = stPar.dir * abs(inst.options.deltaX);
stPar.t1set  = 0; /* [ms] */
stPar.t2set  = 0;
```

2.4 Programming Techniques

2.4.1 GOTO statement

You should **not** use the GOTO statement because it will prevent you from clearly structuring your code.

Should you feel tempted to include a GOTO into your code think of Edsger W. Dijkstra's famous classic paper 'Go To Statement Considered Harmful' published in 1968 (http://www.acm.org/classics/oct95/):

"For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of GOTO statements in the programs they produce."

We have nothing to add to Dijkstra.

2.4.2 Usage of Standard Algorithms

If you need to include a standard algorithm (e.g. for ring buffers, sorting, searching, etc.) don't implement it yourself. Most likely your implementation will not be bug free without some time invested in testing and debugging.

The better way is to copy it in electronic form from a trusted source (e.g. CDs that come with standard text books).

2.4.3 Usage of IEC Data Types

For consistency and target independent code, use the IEC data types in C source code. They are automatically defined with the following statement:

```
#include <bur\plctypes.h>
```

2.4.4 Handling Hardware-Connected (I/0) Variables

It is a good idea to copy (and scale or negate if desired) hardware-connected variables to/from data structures of your software modules in a special task which does just that and nothing else.

You can easily disable this task and disconnect all I/Os for testing purposes. As another benefit you will only have to make minor changes at one single place in the code if external sensor or actor logic changes (which happens quite frequently).

2.4.5 Dynamic Memory Management

Please take care when using dynamically allocated memory. Access to memory which you have not properly allocated leads to errors which are **really hard** to discover!

Don't allocate and free memory frequently in cyclic code because it will lead to memory fragmentation. As a consequence the system will sometimes run out of memory.

2.4.6 Communication between Software Modules

Inter-module communication has to be implemented with global data structures. Therefore it needs to be designed with special care. Please mind appropriate naming of your communication data structures.

2.4.7 Compiler Warnings

Compiler warnings may indicate some unexpected program behavior. Be sure to understand the warning message and correct your code to avoid the warning. If this is not possible or not intended, document the warning at the corresponding line of code.

2.4.8 Determining Array Size

If you need to determine the size of an array (e.g. if you need to check the last array element or need to loop over all elements) use the sizeof function:

```
for (i = 0; i < (sizeof(array)/sizeof(array[0])); i++)
{
  loop body
}
```

2.4.9 Data Alignment

When defining user-defined data types you should note data alignment: in general the compiler has to add empty storage (typically 1 – 3 bytes) between structure members to place (or 'align') the members to specific (e.g. even) memory addresses for memory access.

The actual compiled data size is then larger than the sum of individual member sizes. You can easily check the compiled size with the sizeof function. It may be different for different target hardware architectures.

If you have to write platform-independent code take data alignment into consideration (especially when using data modules). You may place unused alignment bytes into your data structure by yourself to force identical data layout on all your target hardware:

```
TYPE
    Cutter_type: STRUCT
        speed:     REAL;     (* 4 bytes                *)
        cmdcount:  USINT;    (* 1 byte                 *)
        reserve1:  USINT;    (* alignment              *)
        xPosition: UINT;     (* 2 bytes                *)
        yPosition: UINT;     (* 2 bytes                *)
        reserve2:  USINT;    (* alignment              *)
        reserve3:  USINT;    (* alignment              *)
        cutterTON: TON_type; (* align like 4 byte type *)
    END_STRUCT;
```

Please see the Automation Studio online help for details about compiler data alignment.

2.5 Testing

Software testing is a crucial issue for software quality issue. It ensures that the behavior of the code is compliant to the specifications.

Usually the first task in testing is the definition of test cases on the basis of software specification. Testing of special situations and functionalities (special and corner cases) requires special care, e.g. what happens if an incorrect value is passed to a function (‘An effective way to test code is to exercise it at its natural boundaries.’ – Brian Kernighan, one of the creators of the C language).

If you are working in a project team consider testing your code mutually (the code you create is tested by one of your colleagues as an independent tester and vice versa).

Automation Studio™ provides excellent features for software testing: watching, tracing and forcing of variables. These diagnostic methods are extensively discussed in B&R Training Module ‘Automation Studio Diagnostics’.

2.5.1 Unit Testing

The goal of unit testing is to show that isolated individual parts (libraries, modules, functions, ...) are correct.

2.5.2 Integration Testing

In integration testing, individual software modules are combined and tested as a group to verify if they properly work together.

2.5.3 System Testing

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements (IEEE Standard Computer Dictionary). System testing is typically performed at machine or plant commissioning.

2.5.4 Usability Testing

Usability testing measures how well people can handle the machine or plant you have programmed. Usability testing typically focuses on the HMI.

2.6 Documentation

Documenting the software you have created is an important task. On the one hand it supports the users in working with all the functionalities you have provided for him and on the other hand it provides valuable information for other programmers who have to fix a bug or implement some additional functionality into your code.

Documentation on a software project typically consists of information both inside the source-code listings (the code itself and 'comments') and outside them (typically in the form of separate documents).

2.6.1 Comments

Documentation at code level is always aimed at other developers and not at users.

The main contribution to code-level documentation isn't comments, but good programming style: good code is its own **best** documentation.

However, in every program some comments are necessary to explain things about the code that the code can't say about itself (e.g. high-level and low-level organization of programs).

Types of comments:

- Summary comments: should give an overview and a summary of the program at the beginning of the code (like a preface)
- Intent comments (comments on the code's intent): should explain the purpose of a section of code and operate more at the level of the problem than at the level of the solution (explaining the **why** more than the **how**).
- Marker comments: should mark locations where you suspect a bug may exist or where code improvements are planned. They are useful in the development phase and should not appear in completed code.

What you should document:

- Data types (structures)
- Variable declarations (including physical units if applicable)
- Major steps of your routines
- Limitations of your routines
- Global effects of routines
- Interface assumptions
- Timing issues and memory requirements (if critical)
- Revision history

What not to comment:

- Do not use comments to explain things that are obvious to programmers!
- If your code is too difficult to be understood by others rewrite it!

Remember to keep comments up to date when changing the code!

2.6.2 External Documentation

There are two types of external documentation:

- User documentation: contains all information relevant to users of the software (HMI pages, alarms, errors, etc.)
- Developer documentation: contains information for software programmers (description of software design, flow charts, interfaces, etc.)

2.6.3 Documentation Standards

The American National Standards Institute (ANSI) provides ANSI/ANS 10.3-1995 standard for documentation of engineering and scientific computer software at their website http://www.ansi.org for purchase.

The military standard MIL-STD-498 defines software development and documentation standards and is is approved for use by all departments and agencies of the department of defense of the USA (http://www.pogner.demon.co.uk/mil_498/).

Both standards do not focus on industrial automation application software but may provide some valuable general information.

This is version V1.40 [19/07/05] of the B&R Coding Guidelines.

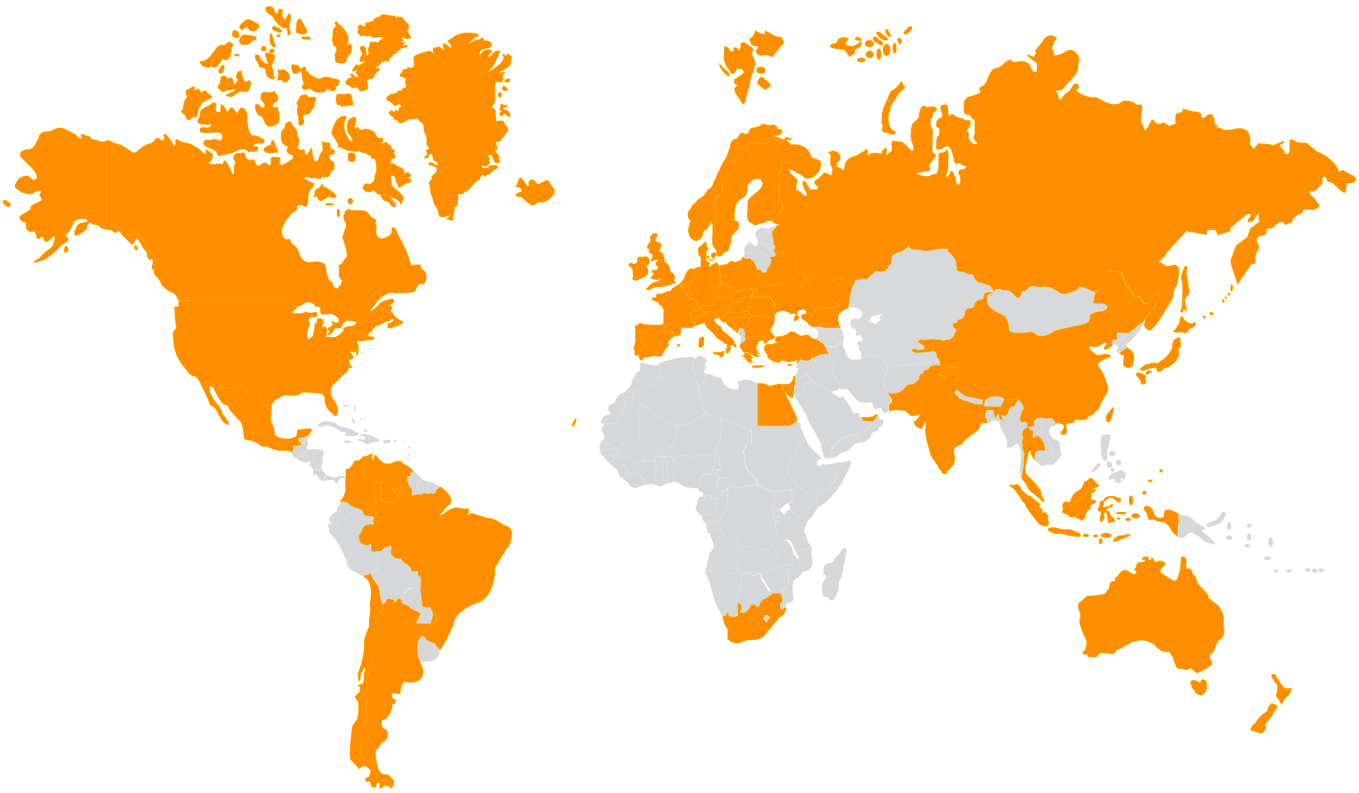
Notes:

Notes:

CORPORATE HEADQUARTERS

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H.
B&R Strasse 1
5142 Eggelsberg
Austria
Tel.: +43 (0) 77 48 / 65 86 - 0
Fax: +43 (0) 77 48 / 65 86 - 26
info@br-automation.com
www.br-automation.com

140 offices in more than 55 countries - **www.br-automation.com/contact**



Australia • Argentina • Austria • Belarus • Belgium • Brazil • Bulgaria • Canada • Chile • China • Colombia • Croatia • Cyprus
Czech Republic • Denmark • Egypt • Emirates • Finland • France • Germany • Greece • Hungary • India • Indonesia
Ireland • Israel • Italy • Japan • Korea • Luxemburg • Kyrgyzstan • Malaysia • Mexico • The Netherlands • New Zealand
Norway • Pakistan • Poland • Portugal • Romania • Russia • Serbia • Singapore • Slovakia • Slovenia • South Africa
Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • Ukraine • United Kingdom • USA • Venezuela • Vietnam