

# TECHNICAL REPORT

**ISO**  
**TR 10562**

First edition  
1995-05-15

---

---

## **Manipulating industrial robots — Intermediate Code for Robots (ICR)**

*Robots manipulateurs industriels — Codification intermédiaire pour robots  
(ICR)*



Reference number  
**ISO TR 10562:1995**  
**Documents**  
Document Sharing Hub

| Contents                                      | Page |
|-----------------------------------------------|------|
| 1 Scope .....                                 | 1    |
| 2 Normative references.....                   | 1    |
| 3 Definitions.....                            | 2    |
| 4 Compliance with this Technical Report ..... | 2    |
| 5 Basic concepts.....                         | 4    |
| 6 Data types.....                             | 13   |
| 7 Instructions.....                           | 21   |
| 8 Data list definition.....                   | 89   |

| Annexes                              |     |
|--------------------------------------|-----|
| A Robot system state variables ..... | 91  |
| B Implementation guide.....          | 92  |
| C National comments.....             | 101 |
| D Code number table of ICR .....     | 109 |

© ISO 1995  
All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Organization for Standardization  
Case postale 56 • CH-1211 Genève 20 • Switzerland  
Printed in Switzerland

## Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The main task of technical committees is to prepare International Standards. In exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

- type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;
- type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;
- type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/TR 10562, which is a Technical Report of type 2, was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC 2, *Robots for manufacturing environment*.

This document is being issued in the type 2 Technical Report series of publications (according to subclause G.4.2.2 of part 1 of the ISO/IEC Directives, 1992) as a "prospective standard for provisional application" in the field of manipulating industrial robots because there is an urgent need for guidance on how standards in this field should be used to meet an identified need.

This document is not to be regarded as an "International Standard". It is proposed for provisional application so that information and experience of its use in practice may be gathered. Comments on the content of this document should be sent to the ISO Central Secretariat.

A review of this type 2 Technical Report will be carried out not later than two years after its publication with the options of: extension for another two years; conversion into an International Standard; or withdrawal.

Annexes A to D of this Technical Report are for information only.

This page intentionally left blank

# Manipulating industrial robots — Intermediate Code for Robots (ICR)

## 1 Scope

This Technical Report specifies an intermediate code to be used as a neutral interface between user-oriented robot programming systems and industrial robot control systems and to define the structure of and the access mechanisms to data lists, which contain data accompanying the intermediate code.

## 2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 8373 : 1994, *Manipulating industrial robots - Vocabulary*.

ISO 8859-1 : 1987, *Information processing - 8-bit single-byte coded graphic character sets - Part 1: Latin alphabet No. 1*.

ISO/IEC 9506-3 : 1991, *Industrial automation systems - Manufacturing message specification - Part 3: Companion standard for robotics*.

### 3 Definitions

For the purposes of this Technical Report, the definitions given in ISO 8373, and the following, apply.

All code and data elements will be represented in accordance with ISO 8859-1 (8-bit character set). ICR is machine or interpreter-oriented instead of user-oriented.

The following words are used in the specialized definition in this Technical Report.

**3.1 ICR program:** Collection of robot directives and data written in ICR.

**3.2 data list:** Collection of positions, poses and other data structures written in ICR codes.

**3.3 robot control unit:** Controlling device which accepts instruction codes and sends servo signals to each joint of an industrial manipulator.

### 4 Compliance with this Technical Report

A program code complies with this Technical Report if it uses only the features of the code that are defined by this Technical Report and it does not rely on any particular interpretation of implementation-dependent features.

The single statements of the Technical Report can be divided into two groups.

The first group (A) contains the statements whose implementation is independent of the industrial robot control and the robot type to be controlled.

The second group (B) contains the statements that are dependent on a specific industrial robot type and robot control.

The group A is divided into 8 compliance levels.

The basic level (level 0) of group A contains:

Memory and data management  
Program flow control  
Boolean and arithmetic operations  
The statements  
    CNFGCHAN, OPENCHAN, CLOSECHAN, RESETCHAN, STATCHAN,  
    DIN and DOUT of the exchange of data statements  
Each addressing mode except symbolic addressing

As extension of the basic level there are 7 compliance levels, which can extend the basic level independent from each other.

|   |                                                                                                           |
|---|-----------------------------------------------------------------------------------------------------------|
| 1 | Symbolic addressing mode                                                                                  |
| 2 | Exchange of data statements except<br>CNFGCHAN, OPENCHAN, CLOSECHAN, RESETCHAN, STATCHAN,<br>DIN and DOUT |
| 3 | Data list management                                                                                      |
| 4 | Sensor functions                                                                                          |
| 5 | Debugging functions                                                                                       |
| 6 | User specific extensions                                                                                  |
| 7 | Future extensions                                                                                         |

The group B is divided into a kernel and an extension.

The kernel of group B contains:

|                                                                     |
|---------------------------------------------------------------------|
| ACCEL, VEL, and MOVTIM statements                                   |
| JMOVE and LMOVE statements                                          |
| CALIB, CURPOS, GOHOME, MOVSTP, MOVCONT, and MOVCANCEL<br>statements |
| Description of end effector facilities                              |

The extension of group B contains:

|                                                                   |
|-------------------------------------------------------------------|
| Each technical specification <u>not contained</u> in the kernel   |
| Each move and effector control <u>not contained</u> in the kernel |
| Description of robot facilities                                   |

An ICR processor is defined by this Technical Report to be "a system or mechanism that accepts an ICR code as input and executes ICR code according to the semantics defined by the standard". A processor complies with the standard if it satisfies at least the basic level of group A and the kernel of group B, except of those functions of the kernel of group B that are explicitly reported by the processor to be not accepted.

ICR may be represented in two different forms.

In the first form all operators only consist of ASCII-character digits. The second form allows mnemonics to express the operators. The effect of the following two statements is exactly the same:

- and
- a) 220, 5450;
  - b) 220, 'MOVEND';

In statement a) the operator is a number, while in b) the operator is in the form of a mnemonic.

Due to this dual representation of ICR-code there are two different kinds of ICR-interpreters. The first and simpler kind needs numerically represented operators, while the second kind also allows using mnemonics. This second kind, called MNEMONICAL-ICR-interpreter, may also be interpreted as a kind of preprocessor to convert mnemonics into numerical values which are reprocessed by a simple ICR-interpreter of the first form.

## 5 Basic concepts

### 5.1. Introduction

ISO/TR 10562 is part of a series of international standards dealing with the requirements of manipulating industrial robots. Other documents cover such topics as safety, general characteristics, performance criteria and related testing methods, terminology, and mechanical interfaces. It is noted that these standards are interrelated and also related to other International Standards.

This Technical Report describes a second committee draft for an intermediate code ICR between off-line programming and different robot control units. There are various levels for the representation of robot programming languages. ICR allows the interchange of technological and geometrical data between robot control units also. This does not mean, that other levels of intermediate code are inhibited. This proposal doesn't eliminate another intermediate code between ICR and robot control units. Nevertheless, ICR should be used as a standardized neutral interface between robot programming and robot control units.

Annex A of this Technical Report provides guidance to the robot system state variables, defined in ISO/IEC 9506-3 "Industrial automation systems - Manufacturing specification - Part 3: Companion standard for robotics", which are referred to in ICR.

Annex B of this Technical Report provides information on how to use and implement ICR at the application and how to come from a robot programming language to ICR code.

A manipulating industrial robot has to perform various tasks. The set of motions and auxiliary function instructions which define a specific intended task of the robot system is specified by a task program.

A given application is specified in terms of task description, or in terms of program, which is written in a user-oriented programming language. The syntax and semantic of this user-oriented high-level programming language depends on the type of system programming, e.g. declarative programming, CAD-oriented interactive programming. The program has to be translated into the low level program code of the robot and then loaded into the robot control unit which will execute it.

According to the number of various robot controllers, and to the evolution of user-oriented high-level programming systems, it is necessary to build a bridge between both kinds of languages.

Such a gateway is called **ICR (Intermediate Code for Robots)**. Its goal is to support all the features that a robot control usually performs and all the elements to describe a task.

The ICR code is useable to exchange user programs and data between different robot control units and/or between any robot programming system and control. The code itself consists of a sequence of records representing different instructions. Such an instruction may be a data or type definition also. The instructions include arithmetic and stack operations, which allow an efficient calculation of arithmetic expressions with respect to a postfix notation.

The block structure of a high-level programming language is supported which helps for structured programming. Variables or other programming objects are identified by symbols with respect to their scope in the block structure. For some special use, absolute addresses may be used, e.g. to handle memory mapped I/O directly.

The proposal shows the scope of work and the functions of ICR in detail, but the formal description is performed on syntax resp. mnemotechnical level. The definition of code numbers is an easy mapping to the ICR elements described in this paper.

Some of the used items like pose or orientation are not explained in detail. Their meaning is described in other ISO documents.

The main features of the intermediate code are as follows:

- Intermediate codes: The position of the code is the standard output format from the robot programming systems and the standard input format to the robot control units as illustrated in figure 1.

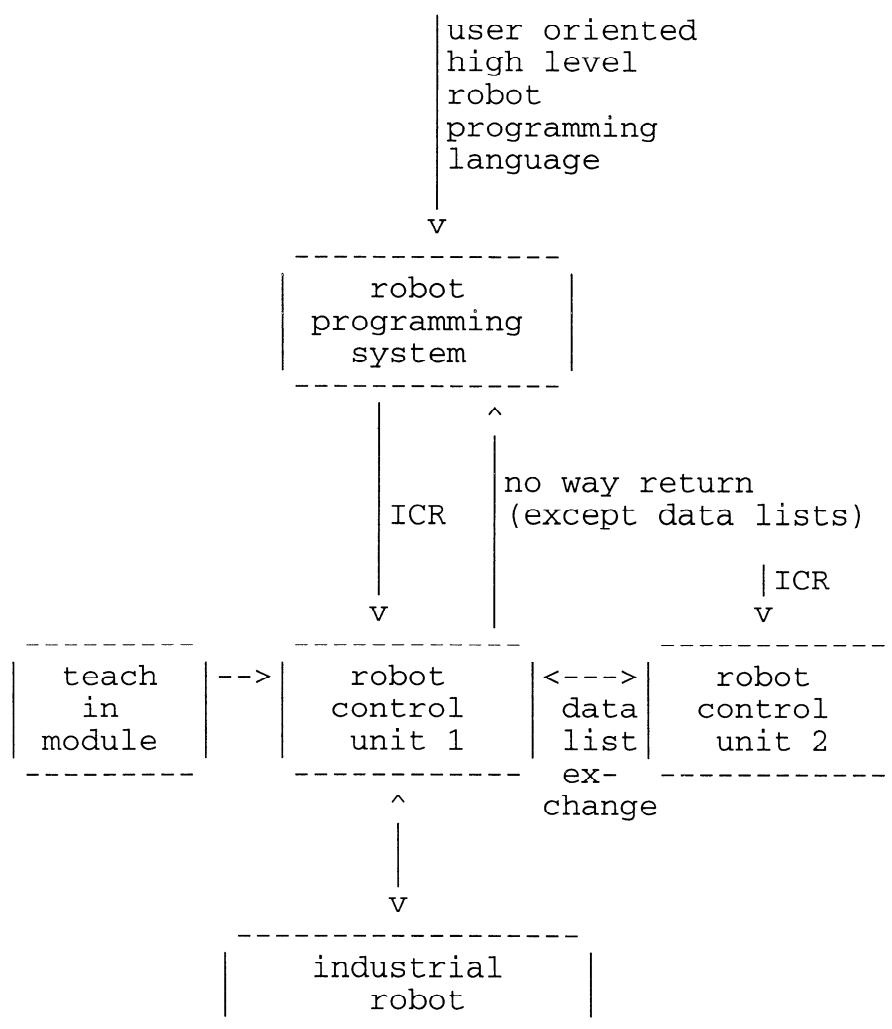


**Figure 1 - Structure of robot programming with neutral interfaces**

- The standard format for data exchange: Data list written in ICR can be used for data exchange and for input data of teaching-by-showing.
- Machine-machine interface: ICR is designed as data exchange format between computer systems. As it is not a user-oriented language, the codes can be represented basically by integer only. However, it uses symbols for variables and others to improve the readability of the program. Printable character set is used in ICR.

Both data and program written in ICR are transferred among the systems in figure 1. The Program is executed in the following two ways:

- a program written in ICR is once converted into that in a user-specified code by means of a translator, and the converted program is transferred to the robot control unit.
- the program written in ICR is transferred to the robot control unit in file or through a communication line, and is directly executed on the robot control unit.

The system configuration and the implementation of a robot control unit is fully dependent on the system developer of the robot. In this Technical Report, however, the robot control unit is assumed to have the structure indicated in figure 2.

- path generator: it computes the trajectory of the industrial manipulator.
- servo system: it realizes the path through servo system.
- peripheral controller: it controls digital/analog input/output units.
- sensors: no specified devices are defined. Instead, general DIO is assumed.
- communication line: a serial line such as RS232C is assumed.
- file storage system: files can be used as option.

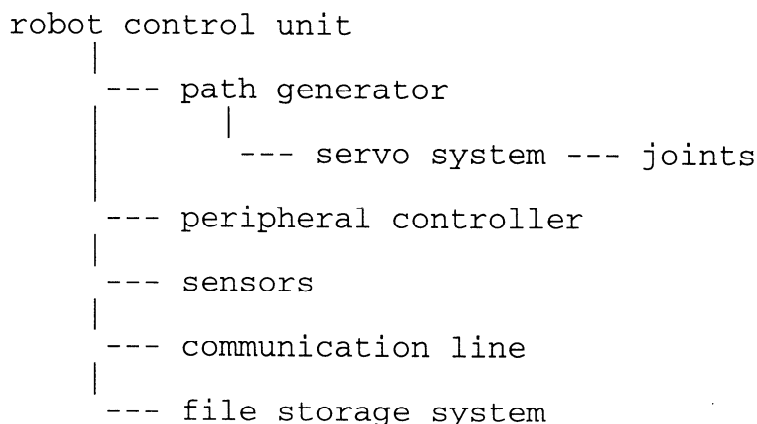


**Figure 2 - Configuration of a robot control unit**

As for an industrial manipulator, an articulated robot with 6 degrees of freedom is assumed. Any robot with redundant degrees of freedom is not covered by this Technical Report. When a robot with less than 6 degrees of freedom is used, it is defined in the same manner as well.

## 5.2 Description of ICR grammar

At the lexical level, an ICR program consists of a sequence of characters of the ISO 8859-1 (8-bit character set) norm. All control characters (such as a linefeed or a carriage return) are ignored in processing.

### 5.2.1 Metalanguage of ICR

The metalanguage for the lexical definition of ICR is derived from the BNF notation:

- terminal symbols (here characters) are enclosed in quotation marks ('');
- parentheses '(' and ')' are used to show grouping;
- concatenation is indicated by the juxtaposition of (terminal or non-terminal) symbols;
- optional symbols are indicated by the square brackets '[' and ']';
- repetition is indicated by the braces '{' and '}';  
*{item1}* means: *item1* occurs not, one time, two times, ... ,  
or n-times
- alternation is indicated by the stroke '|'  
*item1 | item2* means: *item1* or *item2* but not both

Example:

```
number_constant ::= '#' (integer | real) .
```

A number constant can be #30 to define the integer 30 or #0.3E2 to define the real number 30.0.

### 5.2.2 ICR lexical elements

The syntax in this clause describes the formation of lexical elements out of characters and the separation of these elements from each other. Therefore, the syntax is not of the same kind as that used in the rest of this standard. That means, the ICR lexical elements differ from the syntactical definition because no space character is allowed between the terminal symbols of a repetition, e.g. the definition of an integer. The lexical elements will be referenced by the syntactical definitions in the following chapters.

```

digit ::=
    '0' | '1' | '2' | '3' | '4' |
    '5' | '6' | '7' | '8' | '9' .

upper_case_letter ::=
    'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
    'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
    'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
    'V' | 'W' | 'X' | 'Y' | 'Z' .

lower_case_letter ::=
    'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
    'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
    'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
    'v' | 'w' | 'x' | 'y' | 'z' .

letter ::= upper_case_letter | lower_case_letter .

space ::= ' ' .

underscore ::= '_' .

special_character0 ::=
    '!' | '"' | '#' | '$' | '%' | '&' | ''' | '/' .
    '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' .

special_character1 ::=
    ':' | ';' | '<' | '=' | '>' | '?' | '@' .

special_character2 ::=
    '[' | ']' | '\' | '^' | '`' .

special_character3 ::=
    '{' | '|' | '}' | '~' .

graphic_character ::=
    letter | digit | space | underscore |
    special_character0 | special_character1 |
    special_character2 | special_character3 .

unsigned_integer ::= digit { digit } .

```

The semantic of the data types integer, character, real, boolean, and string is described in subclause 6.2.

```
integer ::= [ ( '+' | '-' ) ] unsigned_integer .
```

```
character ::=  
    ''' graphic_character ''' |  
    '$' unsigned_integer .
```

In the description of character, dollar mark '\$' is used to describe the character code number. The type of the character code number is the ISO 8-bits code 8859-1.

```
real ::= integer '.' [unsigned_integer] ['E' integer] .
```

```
boolean ::= 'T' | 'TRUE' | 'F' | 'FALSE' .
```

A **string** is a series of graphic characters surrounded by double quotation marks '"'. To include a double quotation mark itself in a string, a concatenation of double quotation marks is used such as "abc""def" standing for abc"def.

```
string ::= ' " ' {graphic_character} ' " ' .
```

A **symbol** is formed similar to a string with no restriction on its length in which all characters are either letters, digits or underscore and the first character is a letter. There is no difference between small and capital letters (i.e. ABC and abc refer the same object). A symbol is not surrounded by double quotation marks.

```
symbol ::= letter { ( letter | digit | underscore ) } .
```

## 5.3 Address modes

### 5.3.1 Absolute addresses

When users need to specify absolute addresses of robot control units and other peripherals, they can use integer numbers which means absolute addresses of the devices. The addresses should be accepted by the robot control unit. Note that the interpretation of the absolute address is completely system dependent. Therefore users are strongly requested not to use the absolute addresses in order to make the codes portable.

As mentioned already, the computational environment of a robot control unit is out of the scope of this standard. However, we often need to have the image of the memory space, e.g. I/O mapped peripherals, because ICR is a simple, and rather primitive language. Note that the "absolute addresses" are not in every case real addresses, but virtual addresses of the virtual memory space of the CPU. Its data type is positive INTEGER with no restrictions except the range of INTEGER type. It can be denoted as

```
absolute_address ::= '@' unsigned_integer .
```

where unsigned\_integer represents the address of virtual memory space.

Due to very high robot control system dependency the usage of absolute addresses should be avoided.

### 5.3.2 Block relative addresses

A block relative address refers to a variable which is declared for a program block and its inner blocks only. If the block relative address is given by an integer, its value is calculated by the definition formula

$$2^{24} * \text{block\_nesting\_number} + \text{variable\_index}$$

(Remark: The number  $2^{24}$  was defined with respect to a representation of a block relative address by an integer of 32 bits). The `block_nesting_number` refers to an identical number of block nesting given in the block begin statement (BLKBEG) of a block activated before. If several blocks with the same number of block nesting are activated simultaneously, the block that was activated at last is assumed. The block nesting number starts with zero for the main program.

The `variable_index` refers to an identical block relative index of a variable declared by a declaration of variable statement (DECLVAR) inside the block referred to by the `block_nesting_number`.

If a block relative address is given by its symbol, the outer block or procedure name is noted as a prefix with a point, e.g. `procedure_out.variable_name`. If no block or procedure name is written, the inner block is assumed.

```
blockrel_address ::=
    'BRADR' '(' (integer | [ symbol '.' ] symbol) ')' .
```

### 5.3.3 Access to operands

In ICR instructions, the access to operands is performed by using constants, absolute or block relative addresses, or symbols.

```
operand ::=
    symbol | absolute_address | constant | blockrel_address .
```

Part of the operands are pushed on the stack and the rest of them are described in the statements. The results of the execution of instructions are pushed on the stack, if any. When a symbol appears, it is evaluated as a label, a procedure name, or a variable, according to the definition of the instruction.

Constants of integer type are denoted by "#1012", where '#' is ASCII character. "#1012" is the decimal number of one thousand and twelve.

To indicate the data pushed on and popped from the stack, the following notation is used in the definition of each instruction:

Stack:

```
=> I_n: var_type, .... , I_2:var_type, I_1:var_type
    O_m: var_type, .... , O_2:var_type, O_1:var_type
```

where `I_x` indicates an x-th input value and `O_x` indicates an x-th output value, and `var_type` means the type of data.

The stack is "Last In First Out (LIFO)" memory and the orders of input/output values are illustrated in figure 3. Therefore, I\_1 and O\_1 is located on the top of the stack (TOS).

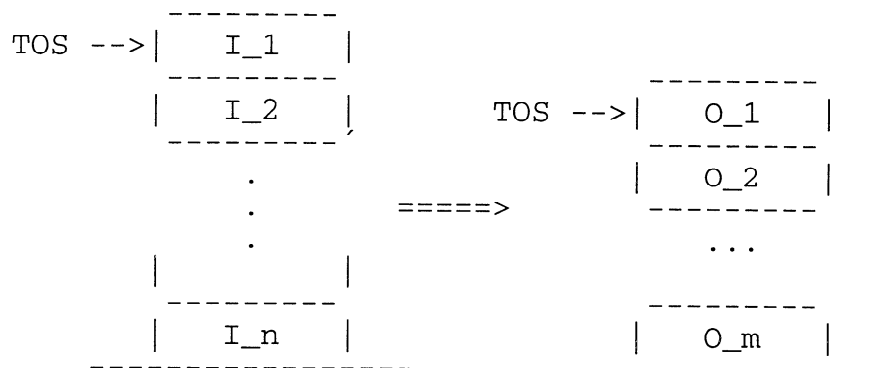


**Figure 3 - Execution of stack operation**

"Var\_type" can be one or more data type names (see "Variables and constants"). When two or more data types are accepted, express them with a pair of parentheses and a vertical stroke as shown in the following way:

`I_1: (var_type1 | var_type2) .`

This notation is out of the lexical definition of ICR, and is only used for the explanation and description of the functionality.

## 5.4 Structure of ICR units

The basic structure of an ICR program consists of units. A unit may include a user program, that can also be a module containing procedures and functions, or a data list including data. Modules are needed to develop program parts separately and link them together. Data lists can be used to exchange data from one robot control unit to another, e.g. the positions for a user program.

```
icr_unit ::= program | data_list .
program ::= pbeg { statement } pend .
data_list ::= dlhead { dldat } dlend .
```

## 5.5 Statement structure

Every operation expressed by ICR is written as a statement consisting of a line number and an instruction. An instruction consists of an operator and zero, one or more operand(s). The operands are separated by a comma and the operation is closed by a semicolon.

All operators are represented by code numbers which are expressed by ASCII-character digits, e.g. '22100'. For the readability of code description, the operators can also be denoted by mnemonics, for example 'PBEG'. Line numbers have to be unique.

```
statement ::= line_number ',' instruction ';' .
```

```
line_number ::= unsigned_integer .
```

```
instruction ::=
    remark | declvar | typedef | progflow | operation |
    specification | movecontrol | dataexchange |
    datalistop | description | sensorfunction | debugfunction
```

Instructions are defined in chapter 7, but for convenience, one example is shown:

```
remark ::= cd_remark ',' linenummer ',' text .
cd_remark ::= 'REMARK' | remark_code .
```

```
remark_code    = 1000 (integer)
linenummer     = linenummer of the high level robot
                programming language (integer)
text           = remark (string)
```

where cd\_remark means code of the REMARK instruction, either the mnemonic 'REMARK' or 1000, the instruction code number for REMARK. Both representations (mnemonics and numerical codes) improve the definition of two classes of conformance for the ICR loader: conformance class 1 adopts codes only, and conformance class 2 adopts codes and mnemonics. "Linenummer" and "text" are the operands of the REMARK instruction.

## 5.6 Program flow structure

As ICR is designed as intermediate code between two computer systems, it does not include complicated program flow structures. Instead, a simple and quick mechanism is introduced such as unconditional branch and conditional branch.

All statements are numbered by serial instruction number which is represented in unsigned integer. The destination of the branch is defined by the serial instruction number, thus

```
label ::= unsigned_integer .
```

It is prohibited to jump into the blocks or procedures from the outside of them.

Block structure is introduced in ICR. In a block, dynamic variables can be used, which must be declared at the head of the block "blkbeg" and are cancelled when the program flow exits from the block. Nesting and other feature of the block are designed in the same manner as those in Pascal.

## 6 Data types

### 6.1 Data processing in ICR

ICR is a code system which computes data in memory, gets information through peripherals and sensors, and controls peripherals and industrial manipulators. Numerical calculation is executed on a stack.

The definition of ICR doesn't include any restriction on memory addressing or scope rules or nesting of procedures and functions. Any restriction caused by implementation must be communicated to users.

#### 6.1.1 Data types

Data types defined in ICR are classified into

- logical and arithmetic data types: integer, Boolean, real, character
- text data types: string
- structured data types: array, record
- geometrical data types: position, orientation, pose
- robot dependent data types: joint, main\_joint, add\_joint, robtarget

The details will be given in subclauses 6.2 to 6.5. Except for the latter four data types, all the other data types have the equivalent data structure in Pascal. Implementation of data structure is system dependent. The latter four types are specially prepared for robot control.

As the mixed computation among different data types is basically prohibited, operations are prepared for each data type and their combinations. Castings of data types are also prepared.

#### 6.1.2 Variables

ICR has two kinds of variables:

- static variable
- dynamic variable

The latter can be used inside of blocks and procedures only.

A variable is represented either by a unique symbol that is declared before use or a variable can be accessed by block relative addressing.

The type of a variable is given by its identifier, which can be a code number or a symbolic name, see the table later on.

`var_type ::= type_identifier | data_type .`

`type_identifier ::= unsigned_integer .`

```

data_type ::=
  'BRADR' | 'INT' | 'REAL' | 'BOOL' | 'CHAR' | 'STR' |
  'POS' | 'ORI' | 'POSE' | 'JOINT' | 'M_JOINT' | 'A_JOINT' |
  'ROBTRT' .

```

Abbreviations of data types are fully described in subclause 7.4.

**Table - Predefined data types in ICR**

| type_identifier | data_type  | data type              |
|-----------------|------------|------------------------|
| 0               | BRADR      | block relative address |
| 1               | INT        | integer                |
| 2               | REAL       | real                   |
| 3               | BOOL       | boolean                |
| 4               | CHAR       | character              |
| 5               | STR        | string                 |
| 6               | (reserved) | (reserved)             |
| 7               | (reserved) | (reserved)             |
| 8               | POS        | position               |
| 9               | ORI        | orientation            |
| 10              | POSE       | pose                   |
| 11              | JOINT      | joint                  |
| 12              | M_JOINT    | main_joint             |
| 13              | A_JOINT    | add_joint              |
| 14              | ROBTRT     | robtarget              |

The data types of the ICR are divided into the following groups:

\* logic and arithmetic data types

They are used for logic and arithmetic calculations like in traditional computer science.

\* text representation

The data type STRING allow to handle text information, e.g. to react on user terminal input or to built-up messages by text blocks.

\* structured data types

Similar to Pascal the user can define structured data types.

\* robot independent geometrical data types

For defining the move of a robot it is very useful to have geometrical data types, which are predefined structured data types. The robot position can be described by POSITION (cartesian coordinates). The robot orientation resp. the end effector orientation can be given by ORIENTATION (cartesian reference coordinate system). A POSE describes the robot position and orientation in cartesian coordinates only.

\* robot dependent geometrical data types

The robot dependent data types can differ from one robot (more precise: robot kinematics) to another. Therefore they are implementation dependent and their definition must be changed, if the ICR-program is transferred from one robot control to another. The axis coordinates are expressed by the data type JOINT. ROBTARGET describes the robot position and orientation in cartesian coordinates like a POSE, but it includes configuration information, used in ambiguous positions, and joint values for additional axis if it is needed.

All datatypes described above are predefined.

## 6.2 Logical and arithmetic data types

For each data type below, a type definition will describe the predefined values which the data type could take. The syntax was defined in subclause 5.2.2.

### BOOLEAN (BOOL)

A data type covering the logical values 'TRUE' and 'FALSE'.

Permitted values are:

false\_value = FALSE | F

true\_value = TRUE | T

### INTEGER (INT)

A data type covering a range of integers.

Permitted values are within a range of integer values where:

lower\_integer\_value = -2 147 483 647

higher\_integer\_value = +2 147 483 647

(The value of an integer variable is represented by a 32-bit word)

### CHARACTER (CHAR)

A data type holding exactly one ASCII character.

Permitted values are:

graphic\_character | range\_of\_unsigned\_integer

range\_of\_unsigned\_character = 0 to 255

The first 128 values correspond to the ASCII-code.

### REAL (REAL)

A data type covering a range of real numbers.

Real number:

real\_value ::= integer '.' [unsigned\_integer] [exponent].

exponent ::= 'E' integer.

A real number is expressed in the conventional notation consisting of a mantissa, and an optional exponent.

The mantissa consists of a positive, null or negative integer value, followed by a mandatory period '.', and optionally by an unsigned integer value expressing the decimal part of the mantissa.

The exponent is a positive, null or negative integer value which represents a power of ten.

No space is allowed within the definition of a real number.

Permitted values are real values within two ranges and true\_zero where:

negative\_range = -1.7E38 to -1.7E-38

positive\_range = 1.7E-38 to 1.7E38

true\_zero = 0.

(The value of a real variable is represented by a 32-bit floating point, see also IEEE 754)

## BRADR (BRADR)

A data type covering a block relative address.

Permitted values are within a range of integer values where:

lowest\_bradr\_value = 0

highest\_bradr\_value = + 4 294 967 296

(A block relative address value is represented by a 32-bit word)

These ranges are minimum requirements for each ICR implementation.

## 6.3 Text representation

### STRING (STR)

A data type consisting of a number of characters enclosed by double quotes ' ' ' and containing information about the actual length of the string. The index of a character in the string starts with 1, the possible length must be at least 255.

## 6.4 Structured data types

There is no explicit data type for structured data types like arrays or records in ICR. Nevertheless it is possible to define arrays in ICR and operate on them by an efficient block relative address administration.

An example should illustrate this. Some ICR-elements used in this example are explained in detail later on. In a potential high level language the following data definitions and assignment are specified:

```
VAR
  X:    RECORD
        K_1: real;
        K_2: integer;
        K_3: array [1 .. 5] of integer;
        K_4: array [1 .. 4] of
              RECORD
                L_1: real;
                L_2: integer;
              ENDRECORD;
        ENDRECORD;
  Y: integer;
```

```
BEGIN
  Y := X.K_4[3].L_2;
ENDPROG.
```

The type declaration can be done as follows:

```
snr, TYPEDEF, 4096, 1, REAL, 1, INT;  
snr, DECLVAR, 0, 1, REAL, 0, 6, INT, 0, 4, 4096, 0, 1, INT;
```

To access the array components for the assignment the following address management by ICR is necessary:

Let 3 be the block relative address of "X":

```
Block relative address of component "X.K_4" -> TOS0:
snr, PUSHBR, BRADR(10);
```

Number of inner record components -> TOS<sub>1</sub>:  
snr, PUSHSZ, 4096;                  number of components  
                                              (L\_1, L\_2)

Index minus lower bound  $\rightarrow$  TOS<sub>2</sub>:

```
snr, PUSHI, #3;
snr, PUSHI, #1;
snr, SUBI;
```

(all 3 lines could be optimized to: "snr, PUSHI, #2")

Check that index is not too large:

```
snr, PUSHI, #3;          TOS3 <- (upb-lwb=4-1=3)
snr, CHECK_BR, error snr, 0;
```

Compute address of indexed element:

```
snr, MULBR;      TOS1 <- TOS2 * TOS1
snr, ADDBR;      TOS0 <- TOS1 + TOS0
```

Compute address of record field "L\_2" -> TOS<sub>0</sub>:

$$\text{snr, ADD\_DIST, BRADR}(10), \text{BRADR}(11);$$

Fetch value and store into variable "Y":

[illegible]

## 6.5 Predefined geometrical data types

## POSITION (POS)

The structured data type for defining the robot position contains 3 real numbers:

```

RECORD   X, Y, Z:   REAL;   END

```

**ORIENTATION (ORI)**

The structured data type for defining the robot resp. end effector orientation with respect to a cartesian coordinate system consists of an integer for indicating the kind of orientation definition and 4 real numbers for the value of orientation (angles or other parameters). The identifier for the kind of orientation definition is used to select the meaning of the 4 reals:

```

RECORD
    ori_id :    INT;
    a,b,c,d :   REAL;
END

```

Examples may be:

```

ori_id = 1
    a, b, c are rotation angles around the x-, y- and z-coordinate axes. d can be any value.
ori_id = 2
    a, b, c are rotation angles around the z-, y'- and z''-coordinate axes. d can be any value.
ori_id = 3
    a, b, c are rotation angles around the z-, y'- and x''-coordinate axes. d can be any value.
ori_id = 4
    a, b, c are rotation angles around the y-, x'- and z''-coordinate axes. d can be any value.
ori_id = 5
    a, b and c are the normalized vector coordinates (for x- and y-direction, and z-axis) of the
    center of rotation. The component d is the rotation angle.
ori_id = 6
    a, b, c, d are parameters of the quaternion specification.

```

**POSE (POSE)**

A pose consists of the position and orientation definition of a cartesian system:

```

RECORD
    POSITION :    POS;
    ORIENTATION : ORI;
END

```

**ROBTARGET (ROBTRT)**

A structured data type for describing a robot move target and different parameters:

```

RECORD
    cartpos      :    POSE;
    (* cartesian position *)
    (* and orientation   *)
    status       :    INT;
    (* information about additional *)
    (* degree of freedom *)
    turns        :    ARRAY [1..r_nturns] of INT;
    (* number of rotations of 360 degrees *)
    adax         :    A_JOINT;
    (* joint position for *)
    (* additional axes   *)
END;            (* RECORD *)

```

Status describes an additional degree of freedom (for example 'shoulder right or left'). The relation between values of data type integer and the robot configuration is system dependent.

Turns describes the number of rotations of 360 degrees performed by some axes. R\_nturns defines the number of joints, which can rotate more than 360 degrees. R\_nturns is robot dependent and fixed and must be known by both the programming system and the robot control system (systemparameter).

Because the number of the additional axes are robot dependent there is a variant record, A\_JOINT, for defining these axis.

#### JOINT (JOINT)

A structured data type for describing a robot destination in joint coordinates. Because the number of axes is robot dependent, joint is composed by the variant records M\_JOINT and A\_JOINT, to distinguish between main axes and additional axes.

```

RECORD
  jointpos :    M_JOINT;
              (* joint position for the *)
              (* main robot axes*)
  adax      :    A_JOINT;
              (* joint position for the*)
              (* additional axes      *)
END;          (* RECORD *)

```

#### MAIN\_JOINT (M\_JOINT)

A structured data type for describing the robot axes in joint coordinates. The data type consists of a number of robot axis values of type REAL. The number of robot axes is robot dependent and fixed and must be known by both the programming system and the robot control system. The number of axes is defined by a system parameter in the control system. According to the axis type, the components are either angles (rotational axis) or distances (translational axis). At the beginning of a program the number of axes can be checked with the CHECK-AXES instruction.

```

RECORD
  A_1, A_2, . . . . :    REAL;
END;

```

#### ADD\_JOINT (A\_JOINT)

A structured data type for describing the additional axes in joint coordinates. The data type consists of a number of axis values of type REAL. The number of additional axes is robot dependent and fixed and must be known by both the programming system and the robot control system. The number of additional axes is defined by a system parameter in the control system. According to the axis type, the components are either angles (rotational axis) or distances (translational axis). At the beginning of a program the number of additional axes can be checked with the CHECK-AXES instruction.

```

RECORD
  AA_1, AA_2, . . . . :    REAL;
END;

```

#### Example:

Assumed an industrial robot has five main axes and two additional axes.

Then the data type M\_JOINT is composed by five reals:

```

RECORD
  A_1, A_2, A_3, A_4, A_5 :    REAL;
END;

```

The data type A\_JOINT is composed by two reals:

```

RECORD
  AA_1, AA_2 :    REAL;
END;

```

And the data type JOINT is composed by M\_JOINT and A\_JOINT:

```
RECORD
    jointpos :    M_JOINT;
    adax      :    A_JOINT;
END;
```

## 6.6 Constants

Constants are denoted by the following definitions:

```
constant ::= simple_constant | structured_constant .

number_constant ::= '#' (integer | real) .

simple_constant ::=
    '#' (integer | real | boolean | character | string) .

structured_constant ::=
    '#' '(' type_identifier ','
    data_structure_definition ')' .

data_structure_definition ::=
    position_cons | orientation_cons | pose_cons |
    joint_cons | robtarget_cons .

position_cons ::=
    number_constant ',' number_constant ','
    number_constant .

orientation_cons ::=
    integer ',' number_constant ',' number_constant ','
    number_constant ',' number_constant .

pose_cons ::=
    number_constant ',' number_constant ',' number_constant
    ',' integer ',' number_constant ',' number_constant ','
    number_constant ',' number_constant .

joint_cons ::=
    main_joint_cons ',' add_joint_cons .

main_joint_cons ::=
    number_constant ',' ... ',' number_constant .

add_joint_cons ::=
    number_constant ',' ... ',' number_constant .

robtarget_cons ::=
    pose_cons ',' integer ','
    integer ',' ... ',' integer ','
    add_joint_cons .
```

## 7 Instructions

ICR consists of a number of instructions, which are ordered by the following groups:

- house-keeping information  
These instructions may be used for transferring additional information to the robot control or especially test facilities. Examples are line numbers and the designation of the user program statements.
- memory and data management  
Instructions for loading and storing of variables and memory allocation.
- program flow control  
To control program execution these instructions include subroutine management, branches, loops and other elements.
- Boolean and arithmetic operations  
These instructions include the boolean and arithmetic operations as they are known by high level programming languages. Additionally arithmetic operations for the geometrical data types are provided, e.g. position transformation by a rotation.
- technical specifications  
Especially for the industrial robot move control technical specifications are needed, e.g. velocity, acceleration, or move time.
- move and end effector control  
These instructions allow the description of robot moves and end effector operations.
- interchange of data  
Instructions Statements for communication with peripheral devices, operator panels or process control.
- data list management  
These instructions statements are for managing external data lists, e.g. including robot positions. They allow operations like read and write list elements, open and close the list file and others.
- description of robot facilities  
There are instructions for limitation of working space, robot kinematics, tool description, units of measurements, length and angles, scaling factors and others.
- end effector control (*not included, future extension*)  
Instructions for end effector (gripper, tools, manufacturing devices) control.
- sensor functions (*not included, future extension*)  
Instructions for sensor specification and control.
- concurrent programming (*not included, future extension*)  
Program elements for task declaration, execution, delay, priorities, and cancelation. Synchronization of tasks or with external events.
- debugging functions (*beginning, future extensions*)  
Commands for testing and special service information, that are not part of a user program in ICR.

- user specific extensions  
The user may specify his own application specific instructions by special instruction notation.
- future extensions  
Code reserved for future extensions.

## 7.1 House-keeping information (REMARK)

This instruction can be used for including remarks identifying the line number in the source program or any other textual information (can be removed during compilation or code loading on control).

Group of compliance: group A, level 0

Syntax:

```
remark ::= cd_remark ',' linewidth ',' text .
cd_remark ::= 'REMARK'| remark_code .

remark_code      =      1000 (integer)
linewidth        =      linewidth of the high level robot
                    programming language (integer)
text             =      remark (string)
```

## 7.2 Memory and data management

### 7.2.1 Declaration of variables (DECLVAR)

Every global or local variable must be declared. The variables are declared by the number of objects and their data type or by their symbolic name and their data type. The actual address of the variable must not be specified.

The block relative indices are given in consecutive increasing order beginning with 0. For structured objects (as well for TYPEDEF objects with a typecode greater than 4095, as for predefined objects) one block relative index is associated for each elementary object, e.g. for four POSITION objects (...4,8,...) 12 indices are associated. The order of the indices is the same as the order of the object declarations through all DECLVAR statements since the last BLKBEG.

Group of compliance: group A, level 0

**Syntax:**

```

declvar ::=
    cd_declvar ',' indicator ',' ((number_of_objects ','
    var_type) | (symbol ',' var_type)) { ',' indicator
    ',' ((number_of_objects ',' var_type) | (symbol ','
    var_type)) } .
cd_declvar ::= 'DECLVAR' | declvar_code
indicator ::= '0' | '1' .
var_type ::= data_type | type_identifier | symbol

declvar_code          =      500 (integer)
indicator              =      indicator for gobal (static) or local
                           variable (integer)
                           0 = gobal
                           (for block nesting equals to 0)
                           1 = local
                           (for block nesting greater than 0)
number_of_objects      =      number of variables of the specified
                           type (integer)
var_type               =      type of the variable
symbol                 =      name of the variable (string)

```

**7.2.2 Type definition (TYPEDEF)**

This statement can be used for defining compound data types. The hereby declared data types can be composed of predefined data types and data types defined by preceding type definition.

Group of compliance: group A, level 0

**Syntax:**

```

typedef ::=
    cd_typedef ',' ( type_identifier | symbol ) ','
    number_of_components ',' comp_type
    { ',' number_of_components ',' comp_type } .
cd_typedef ::= 'TYPEDEF' | typedef_code .

typedef_code           =      700 (integer)
type_identifier         =      identifier of the declared type
                           (integer >= 4096 )
symbol                 =      name of the type (string)
number_of_components    =      number of components of the
                           specified type (integer)
comp_type              =      component data type descriptor
                           (integer)

```

In programming languages type definitions are originally "pseudo codes". Such codes are usually not included in executable code. Due to this understanding ICR may be taken as the transfer representation of source text, not as executable code. These type definitions can also be handled, this means being changed to basic data types, by a preprocessor or a program loader.

### 7.3 Program flow control

These instructions are used to control the execution and the timing of a program. They also include statements for the program and subroutine management.

Syntax:

```

progflow ::=
    pbeg | pend | check_axes | check_ori | check_br | call |
    jump_target | label | subpend | blkbeg | blkend | goto |
    if | jmp_dec | noop | delay | wait_b | wait_le | wait_ge |
    pause | teachon.

```

#### 7.3.1 Program begin (PBEG)

The beginning of a program. This is the first statement of a program.

Group of compliance: group A, level 0

Syntax:

```

pbeg      ::= cd_pbeg ',' first_number ',' name.
cd_pbeg   ::= 'PBEG' | pbeg_code

pbeg_code      = 22100 (integer)
first_number   = number of the first instruction of the
                  programm block (integer)
name           = name of the program (string)

```

#### 7.3.2 Program end (PEND)

The end of a program. This is the last statement of a program. The program execution stops.

Group of compliance: group A, level 0

Syntax:

```

pend      ::= cd_pend.
cd_pend   ::= 'PEND' | pend_code.

pend_code      = 22150 (integer)

```

### 7.3.3 Check\_axes (CHECK\_AXES)

Check the number of axes and the number of additional axes. At the beginning of a program each controller can check if the number of axes and the number of additional axes in this program correspond to the number of axes of the robot and to the number of additional axes. The boolean result is put on the top of the stack.

Group of compliance: group A, level 0

Syntax:

```
check_axes      ::= cd_check_axes ',' ax_number ','
                    add_ax_number .
cd_check_axes ::= 'CHECK_AXES' | check_axes_code .

check_axes_code      =      22850 (integer)
ax_number             =      number of robot axes (integer)
add_ax_number         =      number of additional axes (integer)
```

### 7.3.4 Check\_ori (CHECK\_ORI)

Check the orientation description. At the beginning of a program each controller can check if the orientation description used in this program corresponds to the implemented orientation description of the controller. The boolean result is put on the top of the stack. If several orientation identifiers are used in a program, a check should be done for each orientation identifier.

Group of compliance: group A, level 0

Syntax:

```
check_ori       ::= cd_check_ori ',' ori_id .
cd_check_ori    ::= 'CHECK_ORI' | check_ori_code .

check_ori_code  =      22860 (integer)
ori_id         =      orientation identifier (integer)
```

### 7.3.5 Check value on top of stack (CHECK\_BR)

If the integer value on the top of stack-1 is greater than the value on top of stack, the program execution continues at the instruction defined by instruction\_number. The value on top of stack is removed, the value on top of stack-1 is not removed.

Group of compliance: group A, level 0

Syntax:

```

check_br      ::= check_br_nr | check_br_sym

check_br_nr   ::=      cd_check_br_nr ',' instruction_number
                  ',' block_nesting_difference .
cd_check_br_nr ::= 'CHECK_BR_NR' | check_br_nr_code .

check_br_nr_code      =      22870 (integer)
instruction_number     =      instruction number of the target
                           (integer)
block_nesting_difference =      target block nesting level minus
                           actual block nesting level
                           (integer)

check_br_sym ::=      cd_check_br_sym ',' symbol
                  ',' block_nesting_difference .
cd_check_br_sym ::= 'CHECK_BR_SYM' | check_br_sym_code .

check_br_sym_code      =      22880 (integer)
symbol                 =      label name of the target
                           (string)
block_nesting_difference =      target block nesting level minus
                           actual block nesting level
                           (integer)

```

### 7.3.6 Procedure call (CALL)

The call of a procedure.

The procedure call can be done by using the serial instruction number of the procedure or by a symbol (name of the procedure).

The parameters are passed over the stack.

Group of compliance: group A, level 0

Syntax:

```

call      ::= cd_call ',' subbeg_number .
cd_call   ::= 'CALL' | call_code .

call_code =      22200 (integer)
subbeg_number =      instruction number of the first executable
                    statement of the procedure (integer)

```

### 7.3.7 Subroutine begin or target for a jump (LABEL)

The beginning of a subroutine or a target. The parameters are passed by the stack. The last parameter of a procedure call should be on the top of the stack.

Group of compliance: group A, level 0

Syntax:

```
jump_target      ::= cd_jump_target ',' symbol.
cd_jump_target ::= 'LABEL' | jump_target_code.

jump_target_code = 22400 (integer)
```

### 7.3.8 Subroutine end (SUBPEND)

The end of a subroutine. This is the last statement of a subroutine. A return after the subroutine call statement is executed.

Group of compliance: group A, level 0

Syntax:

```
subpend          ::= cd_subpend .
cd_subpend       ::= 'SUBPEND' | subpend_code .

subpend_code     = 22220 (integer)
```

### 7.3.9 Block begin (BLKBEG)

The beginning of a block. The storage for data objects is prepared. A block is used to restrict the scope of data objects (as defined in a high level robot programming language, e.g. including a Pascal-like data concept) and to manage their memory allocation. Especially for procedures and subroutines the block declaration is useful.

Group of compliance: group A, level 0

Syntax:

```
blkbeg           ::= cd_blkbeg ',' block_num ',' block_nest .
cd_blkbeg        ::= 'BLKBEG' | blkbeg_code .

blkbeg_code      = 22300 (integer)
block_num        = block number (for testing) (integer)
block_nest       = number of block nesting (integer)
```

### 7.3.10 Block end (BLKEND)

The end of a block. The reserved storage for data objects is freed.

Group of compliance: group A, level 0

Syntax:

```
blkend      ::= cd_blkend.
cd_blkend   ::= 'BLKEND' | blkend_code.

blkend_code    =      22310 (integer)
```

### 7.3.11 Unconditional program jump (GOTO)

The program execution continues at the instruction defined by instruction\_number or at the label defined by the LABEL instruction.

Group of compliance: group A, level 0

Syntax:

```
goto ::= goto_nr | goto_symbol .

goto_nr      ::= cd_goto_nr ',' instruction_number,
                  nest_difference .
cd_goto_nr   ::= 'GOTONR' | goto_nr_code .

goto_nr_code    =      22420 (integer)
instruction_number =      instruction number of the target
                        (integer)
nest_difference  =      target block nesting level minus
                        actual block nesting level

goto_symbol     ::= cd_goto_symbol ',' symbol, nest_difference.
cd_goto_symbol  ::= 'GOTOSYM' | goto_symbol_code .

goto_symbol_code    =      22430 (integer)
symbol              =      name of the program part (string)
```

### 7.3.12 Conditional program jump (IF)

If the value on the top of the stack is the boolean value FALSE, the program execution continues at the instruction defined by instruction\_number or at the label defined by the LABEL instruction. The value on the top of the stack is removed.

Group of compliance: group A, level 0

**Syntax:**

```
if ::= if_nr | if_symbol .
```

```
if_nr      ::= cd_if_nr ',' instruction_number .
```

```
cd_if_nr   ::= 'IFNR' | if_nr_code .
```

```
if_nr_code = 22450 (integer)
```

```
instruction_number = instruction number of the first  
instruction after the THEN_program_part  
(if there is an ELSE_part, this will be  
the beginning of the ELSE_program_part,  
otherwise it is the statement to  
continue the program) (integer)
```

```
if_symbol ::= cd_if_symbol ',' symbol .
```

```
cd_if_symbol ::= 'IFSYM' | if_symbol_code .
```

```
if_symbol_code = 22460 (integer)
```

```
symbol = label name of the first instruction  
after the THEN_program_part (if there is  
an ELSE_part, this will be the beginning  
of the ELSE_program_part, otherwise it  
is the statement to continue the  
program) (string)
```

**7.3.13 Jump or decrement (JMP\_DEC)**

Jump on zero or decrement. The program execution continues at the label defined by the LABEL instruction. If the value on the top of the stack is zero, it is removed.

Group of compliance: group A, level 0

**Syntax:**

```
jmp_dec ::= jmp_dec_nr | jmp_dec_symbol .
```

```
jmp_dec_nr      ::= cd_jmp_dec_nr ',' instruction_number .
```

```
cd_jmp_dec_nr   ::= 'JMP_DEC_NR' | jmp_dec_nr_code .
```

```
jmp_dec_nr_code = 22470 (integer)
```

```
instruction_number = instruction number to continue the  
program in case of zero (integer)
```

```
jmp_dec_symbol  ::= cd_jmp_dec_symbol ',' symbol .
```

```
cd_jmp_dec_symbol ::= 'JMP_DEC_SYM' | jmp_dec_symbol_code .
```

```
jmp_dec_symbol_code = 22480 (integer)
```

```
symbol = label name of the program part to  
continue the program in case of  
zero (string)
```

If the integer value on the top of the stack is zero, the program execution continues at the instruction defined by `instruction_number` or at the label defined by the LABEL instruction. Otherwise the integer value on the top of the stack is decremented by one.

### 7.3.14 No operation (NOOP)

No operation is performed. Memory space in the ICR code area will be reserved for testing purposes.

Group of compliance: group A, level 0

Syntax:

```
noop      ::= cd_noop .
cd_noop   ::= 'NOOP' | noop_code .
noop_code      =      22999 (integer)
```

### 7.3.15 Delay (DELAY)

Delay of the program execution.

Group of compliance: group A, level 0

Syntax:

```
delay     ::= cd_delay .
cd_delay  ::= 'DELAY' | delay_code .
delay_code      =      22600 (integer)
```

The program execution is delayed for the time in seconds, according to a real value on the top of the stack. The value on the top of the stack is removed.

### 7.3.16 Wait for binary input (WAIT\_B)

WAIT for binary input or jump if timeout.

Group of compliance: group A, level 0

Syntax:

```
wait_b    ::= wait_b_nr | wait_b_symbol .
wait_b_nr  ::= cd_wait_b_nr ',' instruction_number .
cd_wait_b_nr ::= 'WAIT_BNR' | wait_b_nr_code .
```

```

wait_b_nr_code      =      22620 (integer)
instruction_number   =      instruction number to continue the
                             program in case of timeout (integer)

wait_b_symbol       ::= cd_wait_b_symbol ',' symbol .
cd_wait_b_symbol    ::= 'WAIT_BSYM' | wait_b_symbol_code .

wait_b_symbol_code  =      22630 (integer)
symbol              =      name of the program part to continue
                             the program in case of timeout
                             (string)

```

The program execution is delayed till the binary input is equal to the boolean value on the top of the stack. In case of timeout, the program execution continues at the instruction defined by `instruction_number` or `symbol`. There are three parameters on the stack: `channel_number` (`I_3`, integer), the boolean value (`I_2`), and on the top of the stack the timeout value in seconds (`I_1`, real). If the timeout value is less than zero no timeout check will be done. The values on the top of the stack are removed.

### 7.3.17 Wait for analog input (less than or equal to) (WAIT\_LE)

WAIT till an analog input is less than or equal to a threshold value or jump if timeout.

Group of compliance: group A, level 0

Syntax:

```

wait_le ::= wait_le_nr | wait_le_symbol .

wait_le_nr      ::= cd_wait_le_nr ',' instruction_number .
cd_wait_le_nr   ::= 'WAIT_LENr' | wait_le_nr_code .

wait_le_nr_code    =      22640 (integer)
instruction_number  =      instruction number to continue the
                             program in case of timeout (integer)

wait_le_symbol    ::= cd_wait_le_symbol ',' symbol .
cd_wait_le_symbol ::= 'WAIT_LESYM' | wait_le_symbol_code .

wait_le_symbol_code =      22650 (integer)
symbol              =      name of the program part to continue
                             the program in case of timeout
                             (string)

```

The program execution is delayed till the analog input is less than or equal to the real value on the stack. In case of timeout, the program execution continues at the instruction defined by `instruction_number` or `symbol`. There are three parameters on the stack: `channel_number` (`I_3`, integer), the real value (`I_2`), and on the top of the stack the timeout value in seconds (`I_1`, real). If the timeout value is less than zero no timeout check will be done. The values on the top of the stack are removed.

### 7.3.18 Wait for analog input (greater than or equal to) (WAIT\_GE)

WAIT till an analog input is greater than or equal to a threshold value or jump if timeout.

Group of compliance: group A, level 0

Syntax:

```
wait_ge ::= wait_ge_nr | wait_ge_symbol .

wait_ge_nr      ::= cd_wait_ge_nr ',' instruction_number .
cd_wait_ge_nr   ::= 'WAIT_GENR' | wait_ge_nr_code .

wait_ge_nr_code =      22660 (integer)
instruction_number =  instruction number to continue the
                        program in case of timeout (integer)

wait_ge_symbol   ::= cd_wait_ge_symbol ',' symbol .
cd_wait_ge_symbol ::= 'WAIT_GESYM' | wait_ge_symbol_code .

wait_ge_symbol_code =  22670 (integer)
symbol                =  name of the program part to continue
                        the program in case of timeout
                        (string)
```

The program execution is delayed till the analog input is greater than or equal to the real value on the stack. In case of timeout, the program execution continues at the instruction defined by instruction\_number or symbol. There are three parameters on the stack: channel\_number (I\_3, integer), the real value (I\_2), and on the top of the stack the timeout value in seconds (I\_1, real). If the timeout value is less than zero no timeout check will be done. The values on the top of the stack are removed.

### 7.3.19 Pause (PAUSE)

Pause, wait for start signal.

Group of compliance: group A, level 0

Syntax:

```
pause      ::= cd_pause.
cd_pause   ::= 'PAUSE' | pause_code.

pause_code =      22700 (integer)
```

The automatic program execution is paused and the string on the top of the stack is written to the standard output device. The value on the top of the stack is removed. After the start signal for "program start" the automatic program execution is continued.

### 7.3.20 Teach programming mode on (TEACHON)

The automatic program execution is suspended and the teach programming mode is started. After the end of the teach programming mode and confirmation the program execution is continued.

Group of compliance: group A, level 0

Syntax:

```
teachon      ::= cd_teachon.
cd_teachon   ::= 'TEACHON' | teachon_code.

teachon_code = 22800 (integer)
```

## 7.4 Boolean and arithmetic operations

These instructions are used for boolean and arithmetic operations and for string and joint manipulations. The operands should be on the stack and the last operand is on the top of the stack. The operands are replaced and after the operation the result is on the top of the stack.

Syntax:

```
operation ::=
    ari_op1 | ari_op2 | b_op1 | b_op2 |
    gen_op | str_op | pap_op_c | pap_opv | pap_opvft |
    pap_opvd | pushsz | add_dist .
```

The mnemonic representation of the operation codes consists of a mnemonic code for the operation, followed by one or two letters, indicating the data type of the used operands. Therefore the data types are represented by abbreviations :

| shortest form | short form | data type              |
|---------------|------------|------------------------|
| A             | BRADR      | block relative address |
| B             | BOOL       | boolean                |
| C             | CHAR       | character              |
| I             | INT        | integer                |
| R             | REAL       | real                   |
| P             | POS        | position               |
| O             | ORI        | orientation            |
| E             | POSE       | pose                   |
| T             | ROBTRT     | robtargrt              |
| J             | JOINT      | joint                  |
| M             | M_JOINT    | main_joint             |
| D             | A_JOINT    | add_joint              |
| S             | STR        | string                 |

### 7.4.1 Arithmetic operations

The arguments of trigonometrical functions are assumed to be in radian. In the same way the results of inverse trigonometrical functions are in radian.

#### 7.4.1.1 Arithmetic operations with one operand

Arithmetic operations with one operand are arranged in the following table. The operand should be on the top of the stack (exceptional TOS-1 in case of FLTM1). The operand is replaced and after the operation the result is on the top of the stack (exceptional TOS-1 in case of FLTM1).

Group of compliance: group A, level 0