Table of Content

1. Helmut-Schmidt-University Hamburg

1.1. Overloading

Overloading of Standard Functions is foreseen in the 61131 standard already, however, overloading of user defined functions and function blocks is not foreseen.
This should be added.
This would allow e.g. a transfer Function "To_Real", which can have either "Int" or "Double", etc., as its argument.

1.2. Explicit execution order in Function Block Diagrams

The execution order of the Function Blocks and Functions in a Function Block Diagrams should be explicitly stated, e.g. by means of a "counter" in the FBs and Functions, and should not just be given implicitly by the position of the FB on the screen or by the order in which they have been placed on the screen.
This will help to reduce unexpected effects when an FBD is exported from one tool to another one with a different execution order policy.

1.3. Hierarchical SFC

A clear statement is required how hierarchies in SFC should be implemented, especially whether the super-level SFC has to wait (even if the exit transition condition is fulfilled) until the sub-level SFC has been executed completely, or not.
There are good reasons for both variants, but the execution behaviour is completely different, so we need a clear statement here.

2. Vanstraelen - ST extensions

2.1 Event driven extensions

Two constructions are added: WHEN and CONTROL. These are statements that are not used inside library elements (PROGRAM, FUNCTION, ...) but specified as library elements. They are executed when the value of a variable changes inside another library element.

WHEN

```
WHEN <expression> THEN
        <local variables>
        <instructions>
END_WHEN
```

Every time an element of the <expression> changes it is reevaluated. When a raising edge is detected the <instructions> are evaluated.

Exemple:

```
WHEN pump.temperature > 120 THEN
        pump.run := FALSE;
END_WHEN
```

Nota: a feature that is demanded a lot is a construct CHANGED ... THEN ... END_CHANGED which allows to react on any modification of an expression in stead of only a raising edge.

CONTROL

```
CONTROL <variable> WITH <expression>
```

Every time an element of the <expression> changes it is reevaluated and the assignment <variable> ::= <expression> is executed

Exemple:

```
CONTROL pump.temperature_alarm WITH pump.temperature > 105
```

2.2. Object oriented extensions

To add object oriented features a new library element METHOD is added.

```
METHOD <type> :: <method name> ( <parameter list> )
        <local variables>
        <instructions>
END_METHOD
```

A method describes how something can be executed by a certain type of object.

Inside the definition of the method, the object variable of the type <type> is represented with the identifier **'self'**.
A method does not return a value.
Methods can be redefined for a derived typed.
Methods are virtual. This is important when pointers are introduced.

Examples:

```
METHOD MY_INT::increment()
        self := self + 1;
END_METHOD
```

A method is called with '<-' (arrow)
x <- increment();

Note: the keyword self is borrowed from Smalltalk. The C++ like keyword this would be another candidate but is a pointer to the 'self' object. Pointers don't yet exist in ST, so self seems a better choice.

ALL
ALL <type name> <- <method name> (<parameter value list>)

Executes a method on all variables of a certain type.

Example:
ALL PUMP<-start();

2.3. Combination of object oriented and event driven features

WHEN IN and CONTROL IN

WHEN IN
Define a WHEN or CONTROL for all variables of a certain type

Examples:

```
WHEN IN PUMP self.temperatur > 120 THEN
        self <- stop_running();
END_WHEN
```

CONTROL IN
CONTROL IN PUMP self.temperature_alarm WITH self.temperature > 105

2.4. Pointers and lists

POINTER TO <type> and LIST [<unsigned integer] OF <type>

Dynamic allocation/deallocation was not added to the SCADA system. Garbage collection was not an issue. It would be difficult to maintain the real-time aspects.
To have save pointers every variables has a list of all the pointers that point to it. When leaving the scope of the variable all the pointers are put to NULL.
For a PLC this seems a heavy task. We envisaged 'static pointers'. These are initialized at compile time. They allow complex data structures without the disadvantages of real pointers. Further research on this subject is to be done.

3. Feedback Willi Zeilinger

3.1. Overloaded conversion functions

Overloaded conversion functions (f.e. ' TO_BOOL ' , ' TO_REAL ' ...):

About the usage of overloading and conversion please refer to 2.5.1.4 figures 7 and 9!!!

Another concept could be to improve the concept of the conversions functions and to introduce overloaded conversion functions:

Some advantages:

1.1.This reduces very much the number of conversion functions (from n^2-n to n with n as the number of elementary data types).
It is easier for the customer to find out the correct conversion function.

1.2. The blocks in LD or FBD are much more smaller, the expressions in ST are much more shorter.

1.3. It makes changes of the data types in a POU much more easier:

F.e.:

```
HEADER:
Var1 of type INT
Var2 of type INT
Var3 of type INT

BODY:
Var1:=2;
Var2:=3;
Var3:=REAL_TO_INT(INT_TO_REAL(Var1)/INT_TO_REAL(Var2));
(* Var3 should be 1!!! *)
```

When you change the data types of Var1, Var2 and Var3 to DINT in the header you have to adapt also the body and replace the functions INT_TO_REAL by DINT_TO_REAL and the functions REAL_TO_INT be REAL_TO_DINT.

With overloaded conversion functions you can write:

```
BODY:
Var1:=2;
Var2:=3;
Var3:=_TO_INT(_TO_REAL(Var1)/_TO_REAL(Var2));
```

```
(* Var3 should be 1!!! *)
```

Now you can change the data types of Var1, Var2 and Var3 to DINT without having to adapt the body.
It should also be possible to change the data types of Var1, Var2 and Var3 to REAL. Then the result is Var3 is 0.666.

Also what you can see: the expression is much shorter.

3.2. In ST-editor some additional operations would be very nice:

3.2.1 A continue operator which continues the loop without executing the remaining code

```
for i:= -5 to 5 do
   if i=0 then
      continue
   end_if;
   A[j]:=B[i]/i;
end_for;
```

3.2.2 Unary operators ++,--,+=,-=... like in C:

F.e.:

```
A[i]:=A[i]+5; => A[i]+=5;
A[i]:=A[i]+1; => A[i]++;
```

This simplifies the programming with ST very much and it allows a very effectiv code generation.
So this would also help to save steps in the PLC.

3.3. Additional character literals

In programs with string data calculations often one has a WORD or INT with an ASCII character in it. F.e.

```
if c>=16#30 AND c<=16#39 then (* c>='0' and c<='9' *)
   cNumber:=-16#30+c;
elsif c>=16#41 AND c<=16#46 AND bHex then (* c>='A' and c<='F' *) (* c>='a' and c<='f' *)
   cNumber:=-16#41+10+c;
elsif c=16#23 then (* c='#' *)
```

it would be nice to enter the constants directly as a character literal f.e.

So with f.e. CHAR# as the type literal definition

```
if c>=CHAR#0 AND c<=CHAR#9 then
   cNumber:=c-CHAR#0;
elsif c>=CHAR#A AND c<=CHAR#F AND bHex then
   cNumber:=-CHAR#A+10+c;
elsif c=CHAR## then
```

So with f.e. c# as the type literal definition

```
if c>=c#0 AND c<=c#9 then
   cNumber:=c-c#0;
elsif c>=c#A AND c<=c#F AND bHex then
   cNumber:=-c#A+10+c;
elsif c=c## then
```

additionally one can perhaps take the same escape character as for strings $
so
c#$L: line feed
c#$R: carriage return
c#$0A: line feed
c## or c#$#: #-character

For 16-bit character you could perhaps select f.e. WCHAR# or wc#!!!

3.4. Nested comments or single line comments in ST editor

Problem:
You want to deactivate the following lines

```
(* This is the first comment *)
i:=i+1;
(* This is the second comment *)
i:=i+2;
(* This is the third comment *)
i:=i+3;
(* This is the forth comment *)
i:=i+4;
(* This is the fifth comment *)
i:=i+5;
```

The current solution consists in a lot of single comments:

```
(* This is the first comment *)
(*i:=i+1;*)
(* This is the second comment *)
(*i:=i+2;*)
(* This is the third comment *)
```

```
(*i:=i+3;*)
(* This is the forth comment *)
(*i:=i+4;*)
(* This is the fifth comment *)
(*i:=i+5;*)
```

This is not only very boring but if you want to make it undone you can forget one line and you will get wrong results !!!

Therefore proposal of introduction of nested comments

```
(*
(* This is the first comment *)
i:=i+1;
(* This is the second comment *)
i:=i+2;
(* This is the third comment *)
i:=i+3;
(* This is the forth comment *)
i:=i+4;
(* This is the fifth comment *)
i:=i+5;
*)
```

3.5. Restriction in case_list_element too severe

The restriction that a case_list_element has to be a subrange or a signed_integer or an enumerated_value is too severe: all variables of class VAR_CONSTANT should be allowed too!

F.e. it should be possible to write:

```
MOTOR_OFF as VAR_CONSTANT with 0
MOTER_ON as VAR_CONSTANT with 1

CASE Status OF
MOTOR_OFF:
....
MOTER_ON:
....
END_CASE;
```

3.6. Improvement of the edge detection

At the moment the ladder diagram is the only language which allows simple evaluation of edges without use of function blocks and their instances:

```
  Relay1    Relay2   Relay3
```

```
-----|P|-------|N|---------(S)
```

In all other languages (IL, FBD, ST) you have to use the function blocks and their instances e.g.

```
Header:
VAR R_TRIG_1 R_TRIG
VAR F_TRIG_1 F_TRIG

Body:
R_TRIG_1(IN:=Relay1);
F_TRIG_1(IN:=Relay2);

IF (R_TRIG_1.Q AND F_TRIG_1.Q) THEN
   Relay3:=TRUE;
END_IF;
```

My proposal is to introduce for this reason some special functions with a kind of memory e.g. R_TRIG_FUN and F_TRIG or RTRIG and FTRIG

```
IF (RTRIG(Relay1) AND FTRIG(Relay2)) THEN
   Relay3:=TRUE;
END_IF;
```

This solution is also an improvement for the ladder diagram which also allows solving the following construction very easily:

```
    Relay1  Relay2    +-------------+      Relay3
--------| |-------| |-------|   RTRIG   |-------(S)
                            +-----------+
```

in ST:

```
if (RTRIG(Relay1 AND Relay2) then
   Relay3:=TRUE;
end_if;
```

3.7. REAL Integer Constants

Integer constants should also be valid for REAL numbers e.g. the number '8' should also be valid for REAL (additionally to '8.0').
So a program written for INT or DINT can be easier converted to a program for REAL.

Example:

Header:

VAR Var1 of type INT
VAR Var2 of type INT
Body:
Var1:=Var2/2;

can be changed to

Header:
VAR Var1 of type REAL
VAR Var2 of type REAL
Body:
Var1:=Var2/2;

without any changes in the body.

3.8. Unambiguous definitions of STRING conversion functions

Unambiguous definitions of STRING conversion functions like 'INT_TO_STRING' and 'STRING_TO_INT'

Proposal:
INT_TO_STRING:
Default case with right alignement with leading blanks:
String4:=INT_TO_STRING(5) yields z.B. '...5' (with String4 as STRING(4))

INT_TO_STRING_LEADING_ZEROS:
Special case with right alignement with leading zeros:
String4:=INT_TO_STRING_LEADING_ZEROS(5) yields z.B. '0005' (with String4 as STRING(4))

WORD_TO_STRING:
Default case with right alignment with leading zeros:
String6:=WORD_TO_STRING(16#abcd) yields z.B. '00ABCD' (with String6 as STRING(6))

The advantage is the simple generation of formatted output strings.

Is missing: 'REAL_TO_STRING'!!! Perhaps with additional format parameter!!!

Also the inverse conversion functions 'STRING_TO_INT', 'STRING_TO_REAL' should be defined!!!

3.9. Formatted conversion functions

Formatted conversion functions like the printf-instruction in C should be specified:

Z.B. String1:=FORMAT('Output %2d->%3d : 04x',1,2,16#abc) should yield 'Output _1->__2 : 0abc'!!!

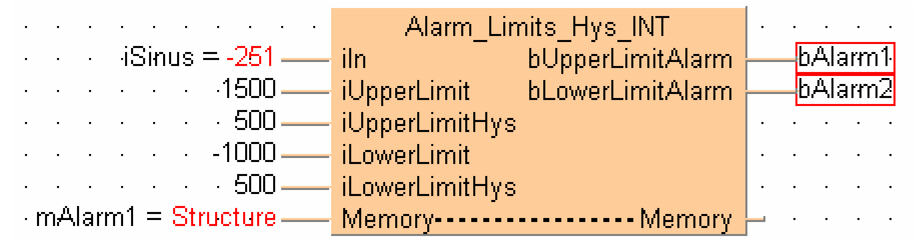
3.10. RTC

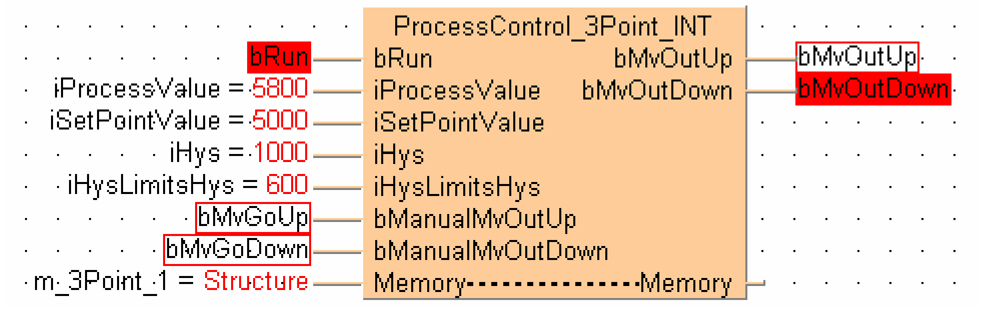
See PLCopen Corrigendum, Jan. 2007

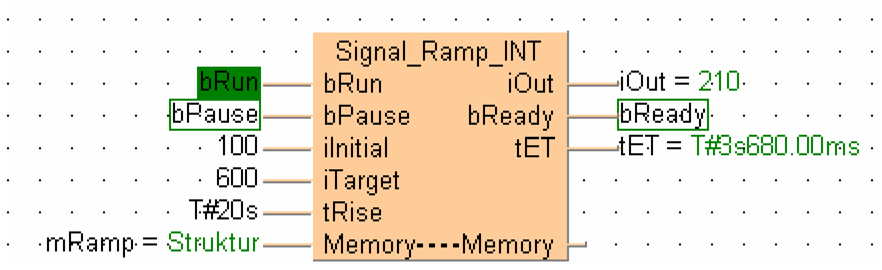
3.11. Functions with result type 'VOID'.

These functions have no result variable with the function name.

If functions have more than one output in general all these outputs must get special names which are different from the function name:
Here some examples:



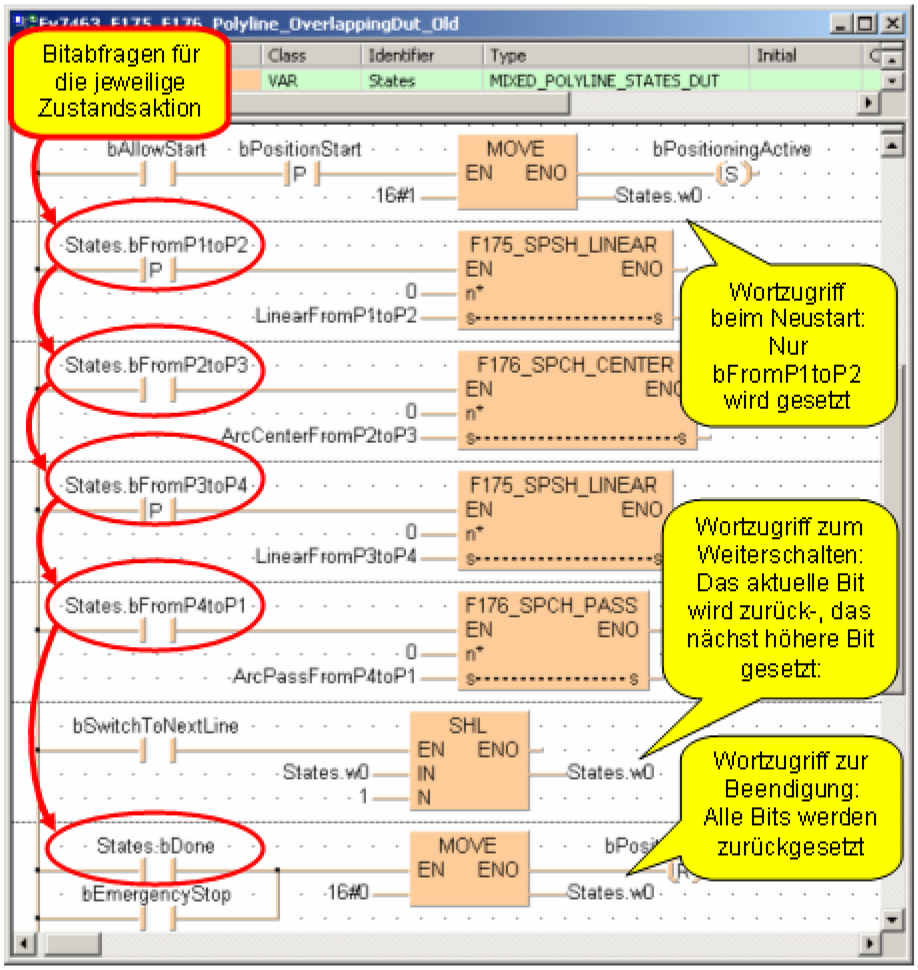




3.12. UNION Statement

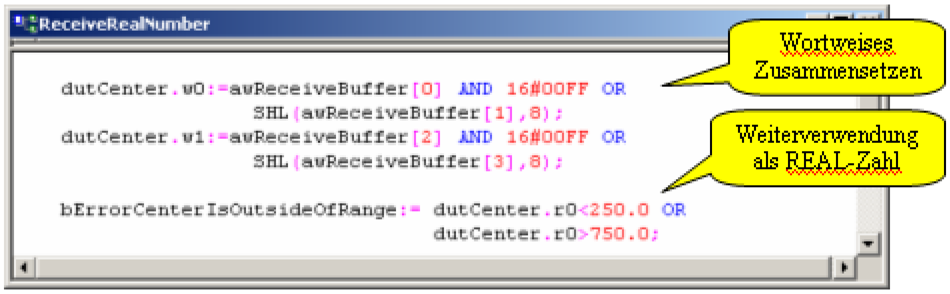
Realization of a kind of union (in C language) solution: access to the same data with different types. In our software a solution is implemented called 'DUT with overlapping elements'.

Examples for this use is mixed bitwise and wordwise access on the same data:



	Class	Identifier	Type	Initial
	VAR	States	MIXED_POLYLINE_STATES_DUT	

or wordwise and realwise access on the same data:



```
dutCenter.w0:=awReceiveBuffer[0] AND 16#00FF OR
                  SHL(awReceiveBuffer[1],8);
dutCenter.w1:=awReceiveBuffer[2] AND 16#00FF OR
                  SHL(awReceiveBuffer[3],8);

bErrorCenterIsOutsideOfRange:= dutCenter.r0<250.0 OR
                               dutCenter.r0>750.0;
```

For the detailed description of these examples and for further information see the article above 'Eine neue Lösung für ein altes Programmierproblem' which was published in the SPS-Magazin 10/2005. <<note editor: not attached here, but available in German language>>

3.13. C editor with special C editor syntax

Define this as a 6th language.

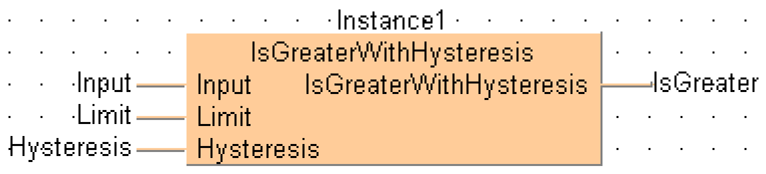
3.14. Use and access of global variables in function:

It is not possible to have access to e.g. large tables which are only available in the global variable list.

3.15. Result variable and result type in Function blocks:

'VOID' means no result variable like proposed for functions.

E.g. the following function:



would be very much easier and more comfortable if the function block had a result type like a function!

3.16. Introduction of VAR_INPUT_TEMP and VAR_OUTPUT_TEMP

In function blocks introduction of VAR_INPUT_TEMP and VAR_OUTPUT_TEMP additionally to VAR_TEMP.

The purpose is to save instance dependent data.

All these proposals eliminate more and more the differences between functions and function blocks. The only remaining difference will be that the data in function blocks (with the exception of the _TEMP variables) have a memory in the function block instances.

3.17. A special function for a detection function of both edges

In all editors: Additionally to the positive and negative edges a special function (in IL, ST, LD, FBD) or a special contact symbol (in LD) for a detection function of both edges

4. Feedback Dieter Hess - Extensions

4.1 Usability of the Standard

- No VAR_EXTERNAL necessary for usage global variables
- Call of PROGRAM from other programs possible (like a FUNCTION_BLOCK with only one instance)
- Mixed data types in expressions; result has biggest type (aReal := bReal + aINT;)
- Assignment with implicit conversions without data loss is possible (aReal := aINT;)

4.2 Less restrictions

- Usage of FBs in structures
- Usage of ARRAYs of FBs
- Complex Datatypes (FBs, ARRAYs) as results of Functions
- VAR_IN_OUT for FUNCTIONs (for performance reasons)
- Access to global variables in FUNCTION_BLOCKs
- Usage of constant expressions in initialisations/Declarations

4.3. Easier programming – Missing features

Bitaccess to WORDs:

```
aWord.7 := TRUE;
```

Single line comment //

Array access to strings (to get the byte value of the chars)

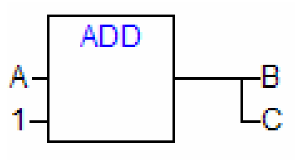
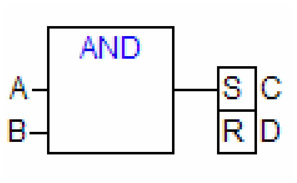
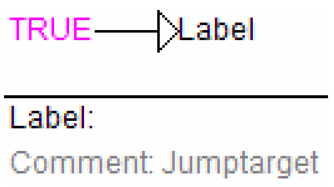
LTIME (time with µs resolution)

CONTINUE in loops (to start at the beginning)

Methods/Flags to control SFC:

- Initialization
- Enable/Disable
- Force Transition
- etc.

4.4. Harmonization of the languages

IL	FBD	ST
LD A ADD 1 ST B ST C		C := B := A + 1;
LD A AND B S C R D		D R= C S= A AND B;
JMP Label: … Label: LD A		JMP Label: … Label: A := B;

4.5 System programming in IEC 61131-3

More and more customers use IEC 61131-3 for system level programming (protocol stacks, IO driver, XML parser, web server, motion control FBs)

Extensions for system programming:
Pointer:

```
A: POINTER TO INT;
B: ARRAY [0..7] OF INT;

A := ADR(B);

A^ := 5;

A[5] := 7;
```

SIZEOF(<Variable>)

Data type **ANY** for user defined FBs

- VAR_INPUT INP: ANY; END_VAR
- Access to INP.SIZE, INP.ADR, INP.TYPE

UNION

4.6. Better support for complex applications

More than one named global variable list

```
MachineState VAR_GLOBAL
        Enable: BOOL;
        Error: BOOL;
END_VAR

FeedUnit VAR_GLOBAL
        Stock: INT;
END_VAR
```

Namespaces for Libraries, Global variables, Enumerations

```
MachineState.Enable := TRUE;
```

References

Like Pointers, but only with explicit assignment

```
        A: REF TO INT;
        B: INT;
        A REF= B;
        A := 5;
```

Preprocessor (conditional compile)

(Editor note: machine appl. program consist of multiple functional units, but not all machnes support these. Via this principle, a subset can be used to compile the runtime)

```
        {define identifier string}
        {undefine identifier}

        {IF expr}
        ...
        {ELSIF expr}
        ...
        {ELSE}
        ...
        {END_IF}

        {include filename}
```

4.7. Better support for complex applications- Object orientation

- FUNCTION_BLOCK can be extended to a class (like C++ extents strcuct to a class)
- METHOD/END_METHOD to add Methods
- Methods called with: Instance.Method(P1, P2, …)
- PROPERTY/END_PROPERTY to define a property (functional access to members)
- EXTENDS to derive from another class

- INTERFACE to define a interface
- IMPLEMENTS to implement a interface within a function block
- THIS/SUPER to access own instance and base class

4.8. Guidelines for extensions

- Extensions can be used in all languages
- Extensions do not replace a IEC 61131-3 feature
- Extensions do not give a IEC 61131-3 feature another meaning
- Extensions are hardware independent

5. Feedback Bob Trask - arrayed instantiation

Arrayed Instantiation is the automatic creation of data memory areas and function block instances (data and algorithm) using arrays.

Example:

```
    VAR
    (* make ten conveyors using the user defined FB_Conveyor *)

    arrConveyors :=  Array[1..10] of FB_Conveyor;

    END_VAR
```

Automatically calling all ten instances of FB_Conveyor:

```
  FOR i := 1 to 10 DO       (* 'i' would be declared as an integer, it is only used to loop the FOR-DO loop
  *)

     arrConveyor[i]();
        (*'call' each of the 1 conveyor instances once every PLC cycle, no data is 'passed' *)

  END_FOR
```

To start the first conveyor all you would need to do is the following (Use of the 'Arrayed Instance' as an 'Object'):

arrConveyor[1].bStart := TRUE;

Nested instantiation:

```
VAR
arrMotor[1..3] of FB_Motor ;       (* declare an array of three motors in FB_Conveyor *)
END_VAR


FOR i:= 1 to 3 DO                  (* within FB_Conveyor, a 'call' of each motor instance *)
   arrMotor(i);
END_FOR


arrConveyor[1].arrMotor[2].bStart := TRUE;  (* 'starting' the second motor in the first conveyor *)
```