Direct/indirect addressing

Addressing methods are mostly tied to variable types, not areas, so the following procedures apply to both DB and Tag variables.

Direct addressing

Direct addressing in Simatic is typically symbolic addressing, meaning in the simplest case we correspond two variables of the same type to each other:

```
fromReal : Real;
fromInt : Int;
toReal : Real;
toInt : Int;
…

#toInt := #fromInt;
#toReal := #fromReal;
```

If the types do not match, conversion will help us:

```
#toInt := REAL_TO_INT(IN := #fromReal);
```

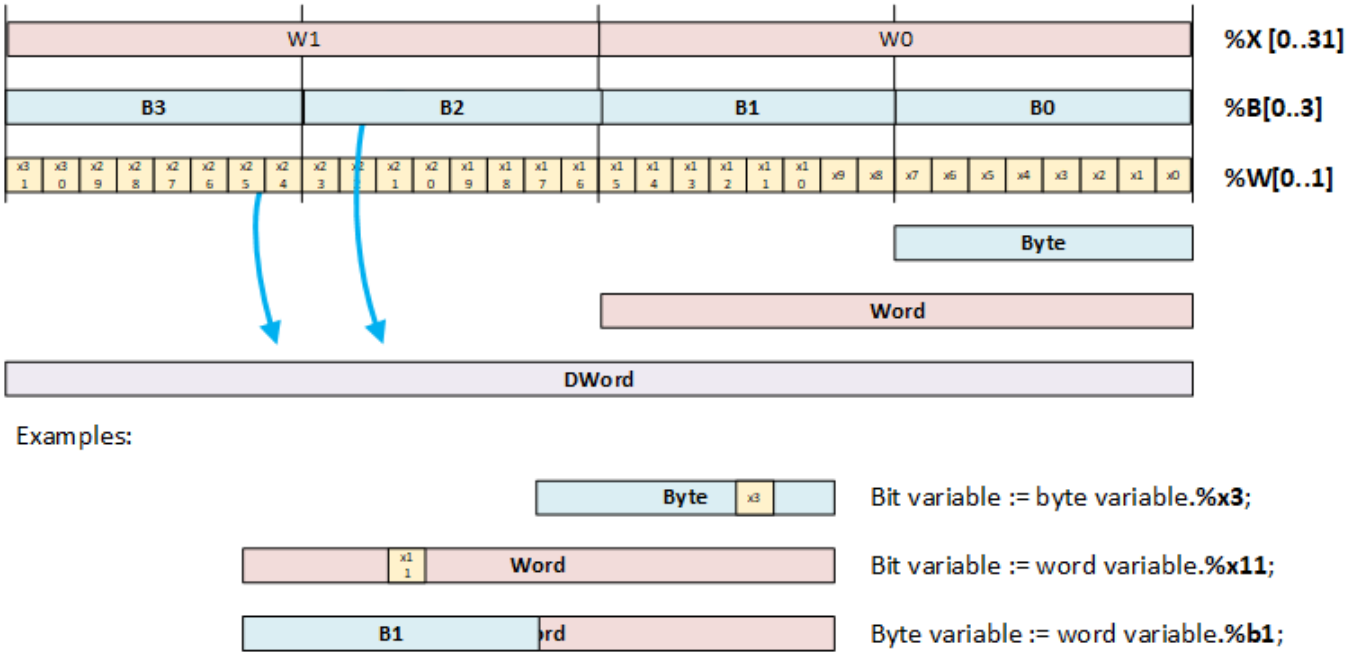
It is crucial to understand that conversion can lead to data loss. In the example above, the REAL type can store much larger numbers and fractional parts, while the INT only handles smaller integers and rounds off fractions. When converting between variables with different ranges, all values outside the smaller range should be considered. In this case, rather than using an INT, a variable with a broader range should be selected (example DINT, LINT).

Direct addressing is also applicable to STRUCTURE and ARRAY types, provided both sides have identical structures.

Another approach is direct addressing, which involves referring to a variable's sub-elements. Although this method applies to a limited range of variables, it is a simple form of assignment. While it isn't as straightforward as the S7-Classic AT command that many programmers prefer, it is at least available:

Slice addressing

Slice addressing involves dividing a memory region, such as a byte or a word, into smaller segments, such as booleans. With S7-1200 and S7-1500, you can target specific parts within declared variables (**only by byte, word, dword**) and access segments of 1, 8, 16, or 32 bits.



The following example is a SPLIT function that splits a WORD Input variable into bits:

```
// FC Input : inWord (Word)
// FC output: 16 variable bit0..bit15 (Bool)
// splitting
#bit0 := #inWord.%X0;
#bit1 := #inWord.%X1;
#bit2 := #inWord.%X2;
#bit3 := #inWord.%X3;
#bit4 := #inWord.%X4;
#bit5 := #inWord.%X5;
#bit6 := #inWord.%X6;
#bit7 := #inWord.%X7;
#bit8 := #inWord.%X8;
#bit9 := #inWord.%X9;
#bitA := #inWord.%X10;
#bitB := #inWord.%X11;
#bitC := #inWord.%X12;
#bitD := #inWord.%X13;
#bitE := #inWord.%X14;
#bitF := #inWord.%X15;
```

Pointer; indirect addressing

In the TIA Portal, there are two ways to perform indirect addressing or pointer referencing: the **ANY** and the **VARIANT**. However, it is important to note that the S7-1200 series PLCs do not support the ANY method. Using a pointer essentially involves moving a data block of a specific size to a memory area of the same size. This operation ignores the structure and variables within the data area, making it a quick and useful method when applied carefully. **However, careless use of this tool can be very risky.**

A key issue is that it doesn't handle the variables within the data being pointed to; for example, when searching for errors with xref, these procedures are not visible to the compiler, which can lead to difficult-to-detect errors caused by improper pointer use.

ANY type

Structure of the ANY Pointer (**10 Bytes**):

Name	Length	Description
Syntax ID	1 byte	Always 16#10 for S7
Data Type	1 byte	Code for the type of data being pointed to (e.g., 16#02 for Byte; see below in the table “TIA Coding of data types”
Repetition Factor	2 bytes	Number of elements of the specified data type
DB Numbe	2 bytes	The number of the data block (0 if not in a DB)
Memory Area	1 byte	Code for the memory area (e.g., 16#84 for DB; see below in the table “TIA Coding of the memory area”)
Address	3 bytes	The start address of the data (bit and byte address)

TIA Coding of data types

The following table lists the coding of data types for the ANY pointer:

Hexadecimal code	Data type	Description
B#16#00	NIL	Null pointer
B#16#01	BOOL	Bits
B#16#02	BYTE	bytes, 8 bits
B#16#03	CHAR	8-bit characters
B#16#04	WORD	16-bit words
B#16#05	INT	16-bit integers
B#16#06	DWORD	32-bit words
B#16#07	DINT	32-bit integers
B#16#08	REAL	32-bit floating-point numbers
B#16#0B	TIME	Time duration
B#16#0C	S5TIME	Time duration
B#16#09	DATE	Date
B#16#0A	TOD	Date and time
B#16#0E	DT	Date and time
B#16#13	STRING	Character string
B#16#17	BLOCK_FB	Function block
B#16#18	BLOCK_FC	Function
B#16#19	BLOCK_DB	Data block
B#16#1A	BLOCK_SDB	System data block
B#16#1C	COUNTER	Counter
B#16#1D	TIMER	Timer

2026/04/23 21:51

TIA Coding of the memory area

The following table lists the coding of the memory areas for the ANY pointer:

Hexadecimal code	Area	Description
B#16#80	P	I/O
B#16#81	I	Memory area of inputs
B#16#82	Q	Memory area of outputs
B#16#83	M	Memory area of bit memory
B#16#84	DBX	Data block
B#16#85	DIX	Instance data block
B#16#86	L	Local data
B#16#87	V	Previous local data

2026/01/16 14:16

Example of using the ANY type

The pointer with the ANY type is most frequently used with the BLKMOV command, which copies data from one area to another indirectly. In the example below, the created dataset is transferred to one of three mobile data areas based on the location of the variable *"assHMI".Panel1ASS* points:

```
CASE "assHMI".Panel1ASS OF
    1:  // ASS1 data to disp 1
        #state := BLKMOV(SRCBLK := P#db108.dbx0.0 BYTE 72, DSTBLK =>
P#db108.dbx216.0 BYTE 72);
    2:  // ASS1 data to disp 2
        #state := BLKMOV(SRCBLK := P#db108.dbx72.0 BYTE 72, DSTBLK =>
P#db108.dbx216.0 BYTE 72);
    3:  // ASS1 data to disp 3
        #state := BLKMOV(SRCBLK := P#db108.dbx144.0 BYTE 72, DSTBLK =>
P#db108.dbx216.0 BYTE 72);
    ELSE  // Statement section ELSE
        ;
END_CASE;
```

VARIANT versus ANY

- **ANY** is a fixed 10-byte structure that references an absolute memory address.
- **VARIANT** is a type-safe pointer that preserves the original data type information and enables symbolic addressing without the need for fixed memory location overhead.
- The **ANY** is an older type, already available in the S300 / 400 series.

* The **ANY** can only be used with the S7-1500, whereas **VARIANT** are accessible on all S7 models.

VARIANT type

A VARIANT type parameter is a pointer that can reference various data types beyond a simple instance. This pointer can be an object of a basic data type like INT or REAL, or it can be a STRING, DTL, ARRAY of STRUCT, UDT, or an ARRAY of UDT. The VARIANT pointer can also recognize structures and point directly to individual members of those structures. An operand of VARIANT type does not consume space in the instance data block or work memory but does require memory on the CPU.

- Directly assigning a tag to a VARIANT, like *myVARIANT := #Variable*, is not possible.
- Direct reading or writing of a signal from an I/O input or output is only possible with an S7-1500 module.
- You can only reference a complete data block if it was originally created from a user-defined data type (UDT).