Memory Management and Data Storage

TM250

Perfection in Automation

www.br-automation.com

Prerequisites

Training modules:	TM213 – Automation Runtime
	TM246 – Structured Text (ST)
Software:	Automation Runtime 2.90
Hardware:	None

Table of contents

1. INTRODUCTION

Working with memory is very important and is a decisive factor when creating professional application software. A solid knowledge of the basics is extremely useful when working with memory in the form of variables, arrays, structures, pointers and freely allocated memory areas, in order to have a clear overview of the entire application.

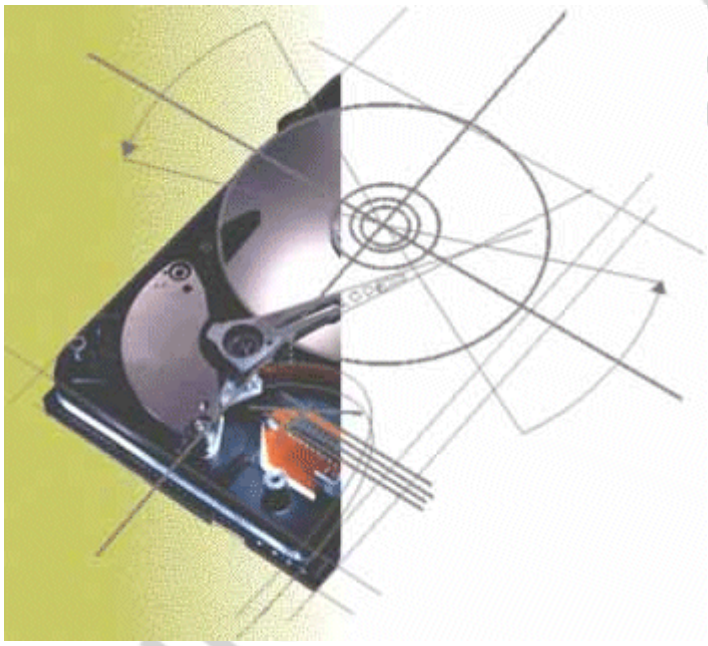


Fig. 1 The parts in a hard drive

In this training module we will cover the basic elements of memory management such as variables, arrays and structures, dynamic variables and memory allocation. We will also take a look at handling data (i.e. creating data objects and files). This training module can be used as a reference tool at any time while you're working. A clear overview of each individual core area is obtained using entry-level subjects and introductions respectively. Examples and exercises will be used to illustrate the practical use of these areas, but are not based on any special programming language.

1.1 Objectives

The course participant will be learning the relationships between variables, arrays and structures and memory allocation.

By the end of the training module, the course participant will also understand the functionality of dynamic variables and dynamic memory allocation.

This training will make the course participant familiar with the options for handling data (e.g. creating data objects and files on the Automation Targets).

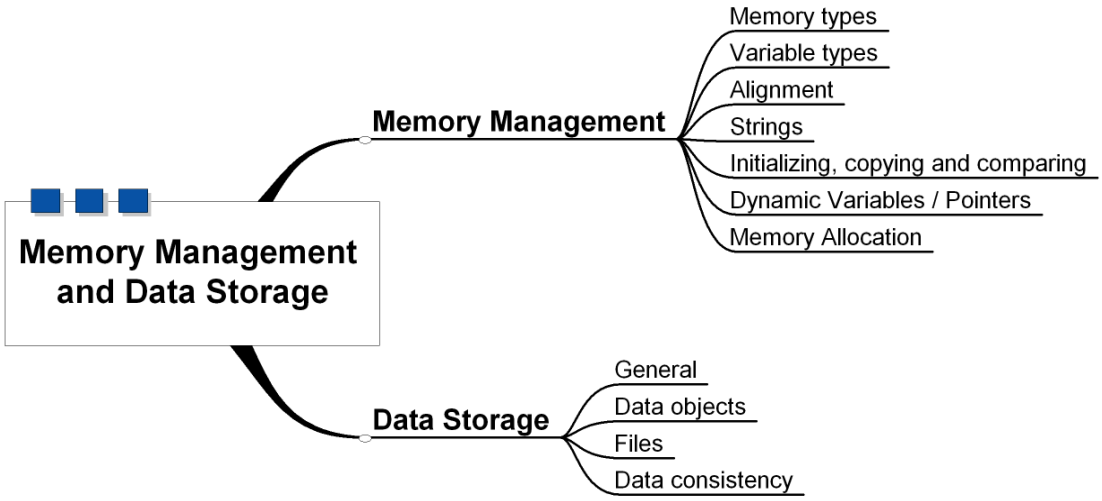


Fig. 2 Overview

2. MEMORY MANAGEMENT

Memory management in the application (whether managed by the operating system or by the user) is a quite extensive topic. The operating system is mostly responsible for managing resources such as time and memory. In open systems, the user is able to manage the memory on his own.
The user can request memory while the system is running, release the memory again or use memory in the form of variables, structures and arrays. However, when doing so, it is particularly important not to lose the overview.
Nowadays, organized and planned memory management has become a more important issue than sparing memory.

2.1 Memory types

In this section will take a detailed look at working with RAM memory, because only this kind of memory is used for all variables and dynamic allocated memory.
Different types of RAM memory have different characteristics regarding data access time and saving.

Memory such as SRAM and DRAM are used:

- SRAM is a static RAM. The data is stored as long these memory elements are supplied with power. A bit set once remains the same until it is reset. When the system is shut off, data is stored using buffer batteries.
- DRAM is a dynamic RAM. Unlike SRAM, this is a high-speed memory type. To store data, DRAM must be refreshed constantly. Data cannot be backed up in DRAM if power is lost. SRAM is used for these requirements.

2.1.1 Access speeds

Access times can vary due to the different characteristics of the memory. In SG4 systems, the SRAM is located on the PCI bus. Therefore, the time needed to transfer data via the PCI bus must also be taken into consideration.

When the system is running, all data is stored on DRAM because of its faster access time. In this case, the SRAM serves as storage medium for remanent data and data objects. Remanent data is the data which must be stored when power is not being supplied or when the system is restarted.

2.1.2 Booting

SYSROM and USRROM are located on the Compact Flash when power is not on. The memory areas REMMEM (remanent data) and USRRAM are located on the SRAM. During booting, the BR objects from the SYSROM and USRROM memory areas are copied into DRAM. REMMEM memory is copied to DRAM in exactly the same way. USRRAM memory is not copied since the amount of time during a power failure is not sufficient to also secure this memory.

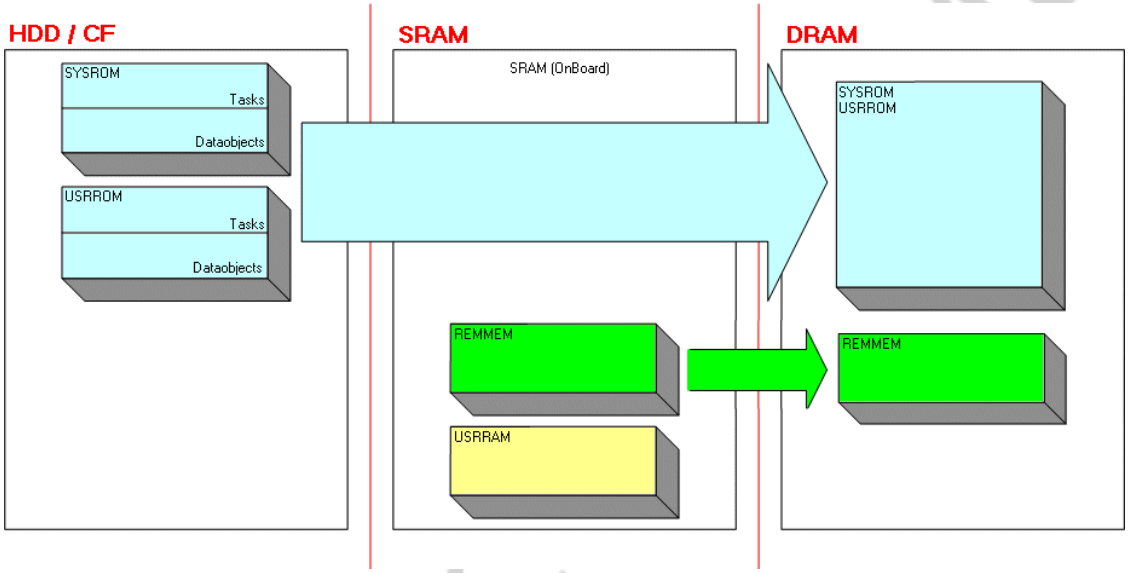


Fig. 3 Booting

2.1.3 Power failure

During a power failure, remanent data from the DRAM is backed up on the SRAM within a limited time frame.

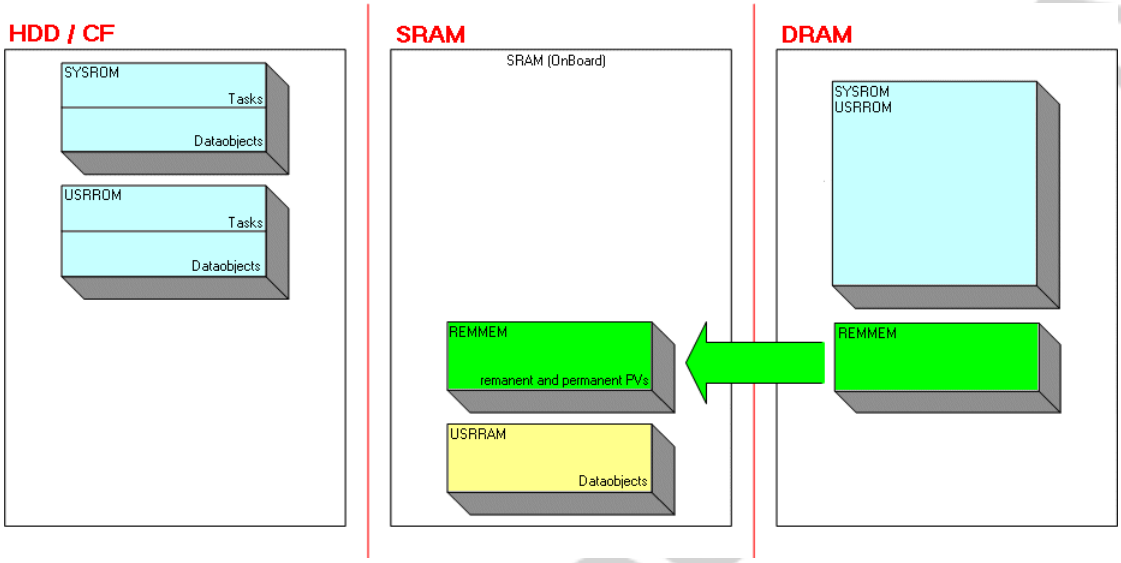


Fig. 4 Power failure

When a power failure is detected, data that should be remanent is backed up in the SRAM.

However, this NMI logic (power failure logic) is not provided on all systems. The corresponding user documentation should state whether the selected system is equipped with NMI logic.

2.2 Variable types

Nowadays, programming is done using symbolic elements with names instead of using fixed memory addresses. These elements are called variables. Simple data types determine the size of the memory, which the variables occupy (value range) and how the contained value is interpreted (with/without sign or decimal point, ASCII text, date/time). An array is a collection of several variables of the same data type, which are addressed using a fixed name and an index. Structures are variables with user-specific data types that the user can create.

2.2.1 Binary and hexadecimal system

To help understand this section, it is useful to have some basic knowledge about the structure of variables in the memory.

A **bit** is the smallest unit of information and can only have the status **0** or **1**. A bit is a BOOL data type.
Other data types consist of multiple bits that can be divided by eight.
The next largest unit is called a byte. A byte is made up of eight bits.

The bits inside a byte are numbered from right to left, from the lowest value **bit 0 to** the higher value **bit 7**.

Bit 2, which is actually the third bit because **counting starts at 0**, has the bit value 4.

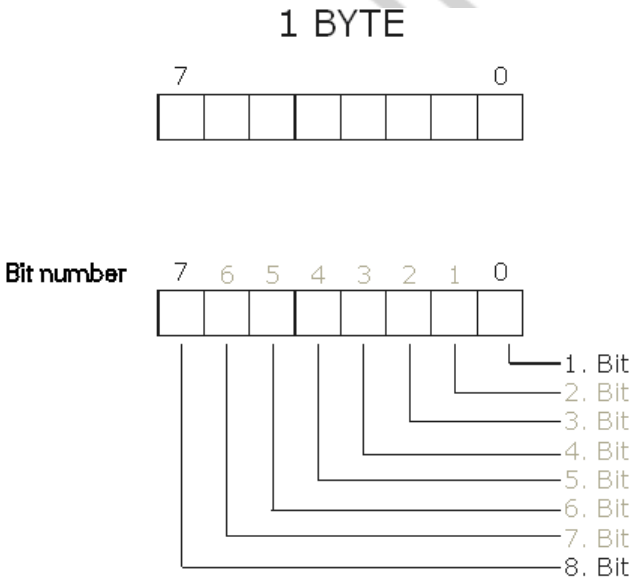


Fig. 5 One byte

A **byte** is divided into two parts, called nibbles or half bytes. The lower nibble is called the low nibble and the higher is called the high nibble. Therefore, the higher value bits are stored in the high nibble.

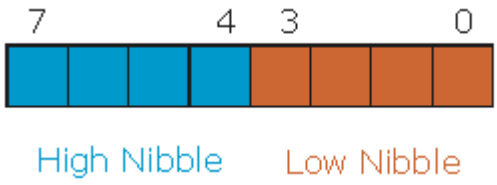


Fig. 6 High nibble and low nibble

Each bit within a byte can have the value 0 or 1. A byte can have the value 0 to 255, which is equal to 256 different states.

The value of the bits is as follows:

7 0	Bit number		$2^{\text{Bit number}}$
00000001	Bit 0 – Bit value	1	2^0
00000010	Bit 1 – Bit value	2	2^1
00000100	Bit 2 – Bit value	4	2^2
00001000	Bit 3 – Bit value	8	2^3
00010000	Bit 4 – Bit value	16	2^4
00100000	Bit 5 – Bit value	32	2^5
01000000	Bit 6 – Bit value	64	2^6
10000000	Bit 7 – Bit value	128	2^7

Example: Adding two binary values

```
 00000001 One
+00000001 One
--------------
=00000010 Two (this is not 10)
```

The following applies:
0 + 0 = 0
0 + 1 = 1
1 + 1 = 0 + carry 1 over to the next bit

In the binary system, **negative numbers** work on the principle that the highest bit is used as the sign. Therefore, 7 bits are left to represent the value.

```
7      0      Bit 7 is the sign
X0000000      Bit 0 to 6 for the value range
X1111111      The largest positive number is decimal 127
```

Negative numbers are made up using the **two's complement**.
The positive decimal number is used to create the bit pattern. This bit pattern is then inverted, and 1 is added. This is how the bit pattern produces a negative number.

Example: Negative numbers in the binary system

```
 00000011    Number is decimal 3
---------------
 11111100    all bits inverted
+00000001    add 1
---------------
=11111101    Number is decimal -3
```

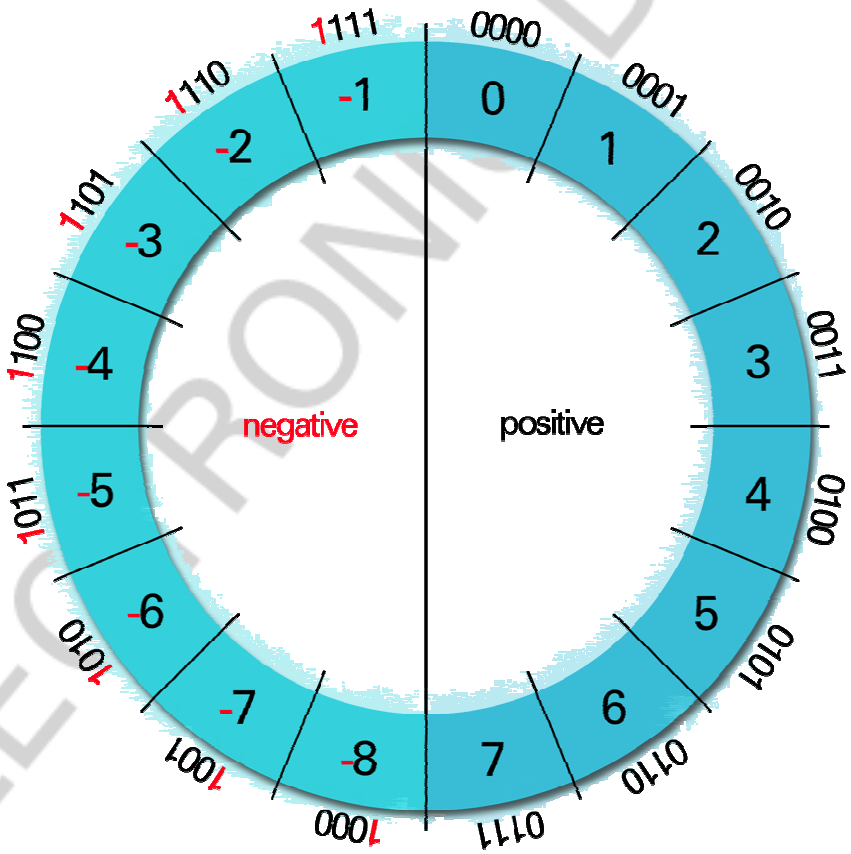


Fig. 7 Two's complement circle

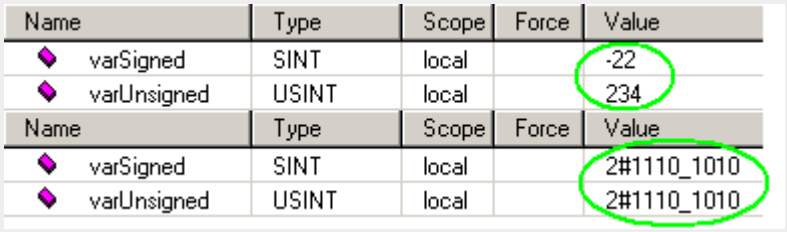
The correct **representation** of the numbers **depends on the data type**.
The bit pattern stays the same if an unsigned data type (USINT, UINT, UDINT) is assigned to a negative value with a signed data type (SINT, INT, DINT). The displayed value however, is different.

Example: Representation of numbers depending on the data type

```
USINT       varUnsigned;
SINT        varSigned;

varSigned   := -22;          (* Bit pattern 1110 1010   *)
varUnsigned := varSigned;    (* Assignment of the value *)
```

The "varUnsigned" variable is displayed as decimal 234, which corresponds to the bit pattern 1110 1010. The bit pattern is not changed. Another value is displayed due to the masking of the data type (unsigned data type).



Name	Type	Scope	Force	Value
varSigned	SINT	local		-22
varUnsigned	USINT	local		234
Name	**Type**	**Scope**	**Force**	**Value**
varSigned	SINT	local		2#1110_1010
varUnsigned	USINT	local		2#1110_1010

Fig. 8 Checking in the Watch window

Unlike the decimal system, 16 values (**0 to F**) are available for one position in the **hexadecimal system**:

```
0000    decimal 0    hex 0
0001    decimal 1    hex 1
0010    decimal 2    hex 2
0011    decimal 3    hex 3
0101    decimal 4    hex 4
0101    decimal 5    hex 5
0110    decimal 6    hex 6
0111    decimal 7    hex 7
1000    decimal 8    hex 8
1001    decimal 9    hex 9
1010    decimal 10   hex A
1011    decimal 11   hex B
1100    decimal 12   hex C
1101    decimal 13   hex D
1110    decimal 14   hex E
1111    decimal 15   hex F
```

Binary numbers are converted to the hexadecimal number system as follows:

Example: Nibble and hex

```
01001011     equal to 75 (64 + 8 + 2 + 1 = 75)
0100         High nibble  = Hex 4
    1011     Low nibble   = Hex B
01001011     Both nibbles = Hex 4B = dec. 75
```

The **nibbles** can simply be **transferred** from the binary **to the hexadecimal** number system and written next to each other.

Example: Hexadizimal und binary Values

```
Binary                                   Hex      Data type
00000000                                $00       1 Byte USINT
00000000 00000000                       $0000     2 Byte UINT
00000000 00000000 00000000 00000000 $00000000 4 Byte UDINT
```

Note:

The hexadecimal representation of numbers is mostly used when displaying **logger entries** and when displaying **addresses**. Comparing bit patterns allows for more effective error searching with **signed and unsigned variables**. The Watch window offers you the option to display variable values in the binary, decimal and hexadecimal system.

2.2.2 Compiler messages

When compiling, the compiler uses different methods to check the program code created by the user. In the message window, errors are displayed as red text and warnings as green text.
The program line causing the error or warning can be displayed by double clicking on the corresponding entry in the message window.

```
* Compiling commx ...
Data type of the local/dynamic (ALIAS)PV > dvar < not given
\pgm\sim\cpu\commx.src:(18) :Error:[Cyclic]  Identifier 'dvar' not defined
\pgm\sim\cpu\commx.src:(20) :Warning:[Cyclic]  '=' signed/unsigned mismatch
1 Error(s) - 1 Warning(s)
```

Fig. 9 Compiler messages

Warnings can occur e.g. when variables with different data types are compared with each other.

Note:

Do not ignore any warnings! Under certain conditions, these warnings can result in application errors during runtime. Warnings should be taken just as seriously as errors.

2.2.3 Simple data types

In programming, there are simple data types also known as basic data types. All available basic data types in accordance with IEC 61131-3 are displayed and categorized in the following list according to their possible areas of use.

Binary	Unsigned	Signed	Floating point	Time, date, string
BOOL	USINT	SINT	REAL	TIME
	UINT	INT		DATE_AND_TIME
	UDINT	DINT		STRING

Data type	Memory requirements [bytes]	Value range
BOOL	1	TRUE (1), FALSE (0) (e.g. digital inputs and outputs)
SINT	1	-128 ... +127
INT	2	-32 768 ... +32 767 (e.g. analog inputs and outputs)
DINT	4	-2 147 483 648 ... +2 147 483 647
USINT	1	0 ... 255
UINT	2	0 ... 65 535
UDINT	4	0 ... 4 294 967 295
REAL	4	-3.4E38 ... +3.4E38
TIME	4	T#-24d_20h_31m_23s_648ms ...T#24d_20h_31m_23s_647ms
DATE_AND_TIME	4	DT#1970-01-01-00:00:00 ... DT#2106-02-07-06:28:15
STRING	Variable, but at least 2 bytes	Character string display

2.2.4 Arrays

Unlike variables, simple data types are arrays of multiple variables of the same data type, which are addressed using a name and an index.

The smallest **index** that can be used to address an array element is 0. The largest index that can be used to address an array element is the total number of elements – 1. This means that the counting order for **index variables** goes from **0 to the number of elements – 1**.

Name	Type	Scope	Force	Value
Pressure	UINT[10]	local		
Pressure[0]	UINT			0
Pressure[1]	UINT			0
Pressure[2]	UINT			0
Pressure[3]	UINT			0
Pressure[4]	UINT			0
Pressure[5]	UINT			0
Pressure[6]	UINT			0
Pressure[7]	UINT			0
Pressure[8]	UINT			0
Pressure[9]	UINT			0

Fig. 10 Array

Access to array elements that are located outside of the valid index range is not monitored by the compiler. This results in memory areas that are located here being overwritten without permission or data being read from an incorrect location.

Access to array elements can look like this:

Example: Accessing array elements

```
UINT          Pressure[10];
USINT         index;

index          := 4;
Pressure[0]    := 3; (*Access to the first array element*)
Pressure[index]:= 22;(*Access via index variable        *)
Pressure[9]    := 12;(*Access to the tenth array element*)
                     (*9 is the highest                *)
                     (*allowed array index             *)
```

In the software, the index variables must be checked carefully to determine whether access to the respective index is even allowed.

Note:

Use the "sizeof" function, to calculate the number of array elements so that the array index can be limited. An example of this can be found later on in this document.

Note:

An index for IEC languages can be **additionally checked** during runtime to see if it has been exceeded using the checking functions described in the Automation Studio online help under **Automation Software:Automation Studio:Programming Languages:IEC Languages:Check Functions**.

2.2.5 Structures

A structure or a user data type is a group of basic data types and/or user data types that are addressed by a common name. Each individual element again has its own name.

Name	Type	Scope	Force	Value
Bread	recipe_typ	global		
flour	SINT			120
water	UINT			12
salt	USINT			1
yeast	UDINT			2

Fig. 11 Structure

Structures are mostly used for grouping data and values that have a common reference to each other. This is similar to a baking recipe, which always uses the same ingredients but in different amounts depending on the recipe.

The individual elements can be accessed as follows:

Example: Accessing elements of a structure

```
recipe_typ    Bread;

Bread.flour   := 120; (* Access to the flour element *)
Bread.water   := 12;  (* Access to the water element *)
Bread.salt    := 1;   (* Access to the salt element  *)
Bread.yeast   := 2;   (* Access to the yeast element *)
```

2.2.6 Arrays of structures

It is also possible to create **arrays of structures**. The sub-elements of the array are addressed via an index the same way as for arrays of basic data types. The same rules apply for the index as for arrays.

Name	Type	Scope	Force	Value
Breads	recipe_typ[3]	global		
Breads[0]	recipe_typ			
flour	SINT			120
water	UINT			12
salt	USINT			1
yeast	UDINT			2
Breads[1]	recipe_typ			
flour	SINT			100
water	UINT			11
salt	USINT			1
yeast	UDINT			1
Breads[2]	recipe_typ			
flour	SINT			88
water	UINT			8
salt	USINT			0
yeast	UDINT			1

Fig. 12 Array of a structure

The sub-elements in arrays of structures are accessed as follows:

Example: Accessing elements in arrays of struktures

```
recipe_typ          Breads[3];

index                 := 1;
Breads[0].flour       := 120;(* Access to array element 0 *)
Breads[index].water   := 11; (* Access via index variable *)
Breads[index].salt    := 1;
Breads[2].yeast       := 1;  (* Maximum array index is 2  *)
```

Here, you should also check whether the index is in the permissible range. This is 0 to 2 in this case.

Note:

Use the "sizeof" function, to calculate the number of array elements so that the array index can be limited. An example of this can be found later on in this document.

2.2.7 Working with the "sizeof" function

When developing software, much importance is placed on writing code that is efficient and easy to work with. However, this is often only possible with data that was determined during runtime.
The information about the size of variables, arrays, structures and arrays of structures can be determined using the "sizeof" function.

This function returns the size of the specified element in bytes.

The "**sizeof**" function is very useful for the following application:

- Determining the size of variables
- Determining the end value for looping to initialize arrays
- Offset calculation

Determining the **size of a variable**:

Example: Using the function sizeof

```
UINT      SizeInByte;
DINT      SetPosition;

SizeInByte:= sizeof(SetPosition);
```

The return value from the function is the number of bytes used in the memory for this variable.

Initializing elements from an array using a **loop**:

Example: Loop with static end value

Method 1: Using numeric constants

```
UINT          Pressure[10];
UINT          lcnt;

Loop lcnt := 0 to 9 do
    Pressure[lcnt] := 0; (* Initializing all elements *)
End_Loop
```

Example: Loop with calculated end value

Method 2: Calculating the end value of the loop with "sizeof"

```
UINT          Pressure[10];
UINT          lcnt;
UINT          szAll;
UINT          szSingle;
UINT          elem;

szAll   := sizeof(Pressure) (*Size of the array in bytes*)
szSingle:= sizeof(Pressure[0])(* Size of an element     *)
elem    := szAll / szSingle
(* Number of elements = All bytes / bytes in an element *)


(* End value of the loop is (elements - 1) !!! *)
Loop lcnt := 0 to (elem - 1) do
  Pressure[lcnt] := 0; (* Initialization of all elements*)
End_Loop
```

Example: Simplified solution

Method 3: Combination of the expressions

```
UINT        Pressure[10];
UINT        lcnt;

Loop lcnt:= 0 to (sizeof(Pressure)/sizeof(Pressure[0])) - 1 do
    Pressure[lcnt] := 0;          (* Initialization of all elements*)
End_Loop
```

Determining the size of a **structure**:

Example: Determining the sizeof of a strukture

```
UINT          SizeInByte;
recipe_typ    Bread;

SizeInByte := sizeof(Bread);
```

Adding together all of the bytes for this data type should equal 8.

Name	Type	Scope	Force	Value
Bread	recipe_typ	global		
flour	SINT			120
water	UINT			12
salt	USINT			1
yeast	UDINT			2

In actuality, a size of 12 bytes was determined by the sizeof function.

Name	Type	Scope	Force	Value
SizeInByte	UINT	local		12

How does this happen?
This happens because of the **alignment behavior** of the processor. The following section will explain this in further detail.

Note:

The use of fixed numbers in the program does shorten the program code, however:
If e.g. the size of the array changes, then changes must also be made in the program code. The more complex method of writing the code does not have this problem. The code becomes more **efficient** and **dynamic**. The **code does not have to be changed** when the array sizes change. Therefore, nothing can be overseen.

2.3 Alignment

The size of structures in the memory is not necessarily the number of bytes used in the structure.
The alignment is responsible for this. It makes sure that variables with 2 or 4 bytes are only placed on addresses that can be divided by this value. However, this depends on the processor being used.

In this case 8 bytes should actually be used according to the structure of the data type.

Name	Type	Scope	Force	Value
Bread	recipe_typ	global		
flour	SINT			120
water	UINT			12
salt	USINT			1
yeast	UDINT			2

Fig. 13 Structure

In actuality, a size of 12 bytes was determined by the **sizeof** function.

Name	Type	Scope	Force	Value
SizeInByte	UINT	local		12

Fig. 14 Structure size

What is behind the alignment?
This mostly depends on the processor architecture. The respective processor type has addressing rules that the compiler must adhere to.

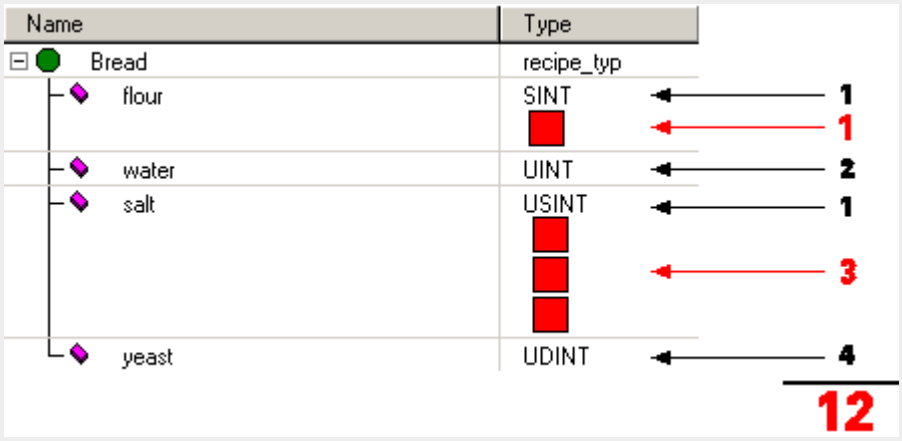
In principle, no variables can begin at an **odd** memory address.

In byte arrays, the array starts at an **even** address, but the individual bytes can be located alternately on even and odd addresses.

Variables and elements, whose data type requires two bytes in the memory can only ever be located at an address that is divisible by two (i.e. an even address).
Four byte values in SG4 systems can only be located at addresses divisible by four.

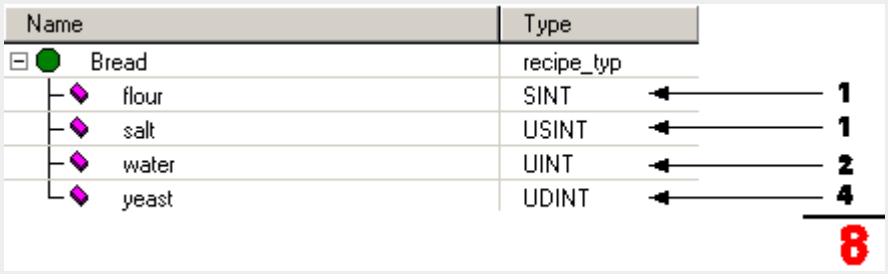
Example: Fillbytes in a struktue

The number of bytes used in the structure adds up to 8. The compiler inserts a filler byte after the first byte. The element "water" then begins at an even address again. 3 filler bytes are inserted after the element "salt" so that the last element, whose data type is UDINT, can begin at an address divisible by four.



Altogether, a total of 12 bytes are used when taking the alignment into consideration.

The filler bytes can be used when changing the order of the elements "water" and "salt".



In this case, the compiler does not have to insert any filler bytes because each element starts at a memory address at which it can also be located.

The alignment is a little different depending on the platform being used (SG3 or SG4). The alignment described above applies to SG4 systems. The same rules apply to SG3 systems, except that 4 byte values can also start at addresses divisible by two.

Note:

When structuring user data types, it is recommended to use the filler bytes as shown in the example or to fill-out using reserve bytes.

A data type that is structured for alignment on SG4 systems can also be used on SG3 systems without any problems. This means that the software can run the same on both systems.

2.3.1 Determining addresses

Each memory location has its own address. An address is essentially no different than a room number in a hotel. A memory location can be uniquely identified using this address.

The compiler (not the user) assigns addresses and offsets to the variables, constants, arrays, structures and arrays of structures. This means that addresses are always used which were generated by the system and made available to the user.

Addresses are determined as follows:

Example: Determining the address of a variable

```
UDINT     adrCounter;
USINT     Counter;

adrCounter:= adr(Counter);
```

The value returned by the function is the address of the variable in the memory.
The address value is always a UDINT data type!

Note:

When compiling, a variable list is created where each variable is given a specific offset. When booting the controller, the operating system reserves a specific area of the RAM for variables, (i.e. it determines a start address). However, it is not necessary for this to always be the same address. It could be a different address after the next software download or after a system restart.

Therefore, **fixed addresses** and address values that have already been determined once should never be used!

Always use the "adr" function to determine addresses.

2.4 Strings

A string is a byte array, where each contained value (0-255) is interpreted as a character. The ASCII code defines which value represents each character (ASCII = American Standard Code for Information Interchange).
A string must always end with null-termination, which means that **decimal 0** must always be in the byte after the last character (not to be confused with the ASCII character "0", which is actually decimal 48!).
The **null termination** is used so that all string processing functions can recognize the **end of the character string**.
As a result, the length of a string variable must always be 1 byte larger than the maximum number of characters contained.

Caution:

When **declaring** string variables, there are **differences between ANSI C and the other programming languages**:
In ANSI C, the null termination must be taken into consideration when declaring (i.e. a STRING[10] can contain a maximum of 9 characters plus the null termination).
Null termination is added automatically in the other programming languages. If, e.g. a STRING[10] is defined, it can really contain 10 characters because the string variable is automatically enlarged to 11 bytes (10 characters plus null termination).
This behavior is easily checked using the "sizeof" function!

A string is structured as follows.

The string variable shown in the image occupies 10 bytes in the memory. Therefore, it can contain a maximum of 9 characters and the null termination.

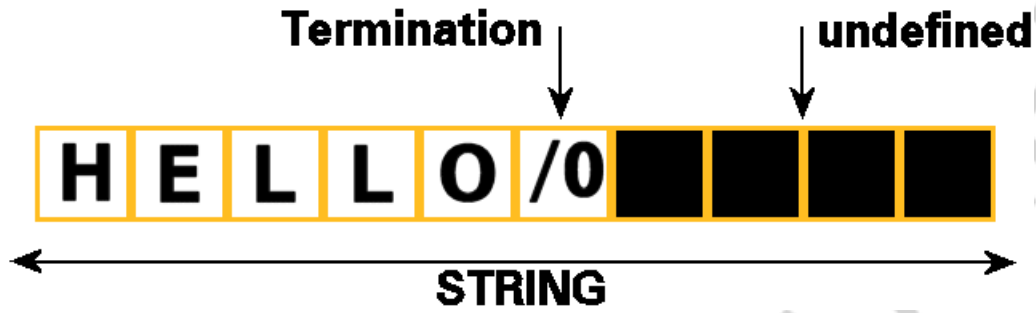


Fig. 15 String structure

The content of the bytes **after the null termination** is **undefined** as long as all possible positions in the string have been used.

2.4.1 Copying strings

When copying strings, the address of the source string and the destination string is always specified. The character string including the null termination of the source string is copied.
Make sure that the destination string is at least exactly as long as the source string. Otherwise, data located after the destination string could be mistakenly overwritten.

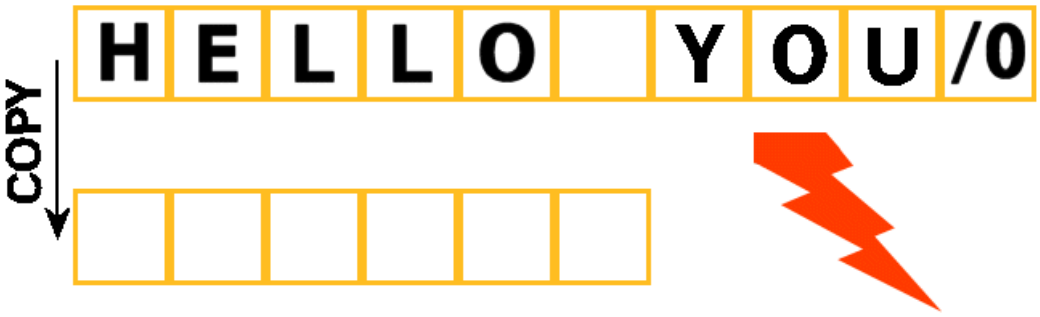


Fig. 16 The source string is longer than the destination string

Example: Using the function strcpy

```
STRING    strSource[10];
STRING    strDest[10];

strcpy(adr(strDest), adr(strSource));(* Copying process *)
```

The address of the destination string is specified first, then the address of the source string.
The "strcpy" function is contained in the "AsString" library.

2.4.2 Comparing strings

It is often necessary to compare two character strings with each other. The "strcmp" function can be used to do this.

Example: Comparing two strings

```
STRING    strSource[10];
STRING    strDest[10];
DINT      status;

status := strcmp(adr(strDest), adr(strSource));

if status = 0 then
    (* Both strings are the same *)
else
    (* The strings are different *)
end_if
```

Case is also taken into account in the comparison. Therefore, the binary code for both strings must be identical for the function to return the status = 0.

2.4.3 Attaching strings to one another

The "strcat" function can be used to attach several strings together. The respective substring is attached to the destination string.

Example: Attaching strings with strcat

```
STRING      strSource[4];
STRING      strDest[10];

strcat(adr(strDest), adr(strSource));
```

If "strDest" has the content "**ABC**" and strSource the content "**def**" before the function is called, then the substring is attached when the "strDest" function is called.
This results in "**ABCdef**".

The **length of the destination string** must be large enough to save both substrings!

2.5 Initializing, copying and comparing

In some cases, memory areas must be initialized, copied to another location or compared with each other because of the structure of programs or how the tasks are handled in the application.

An initialization value can be specified for variables at the time of declaration. The init subprogram in each task can be used for a one-time initialization of variables when booting the controller.

Functions for initializing, copying and comparing memory areas from the "AsString" library will be described in more detail in the following sections.

2.5.1 Initializing with "memset"

Applications often require the initialization of defined memory areas. The most common case is a comprehensive initialization of all of the bytes in a memory block with the value 0.
The "memset" function can be used to initialize the bytes in a memory area with a value.

Example: Initializing a certain memory by using memset

```
USINT    Data[10];
memset(adr(Data), 0, sizeof(Data));
```

When the function is executed, the number of defined bytes is set to the specified value beginning at the specified address.

2.5.2 Copying with "memcpy"

The "memcpy" function makes it possible to copy memory blocks.
The function does not check the source address, destination address, or the length of the data. Therefore, the user must make sure that the specified addresses and lengths really are correct in the application.

Example: Unsing the function memcpy

```
USINT    srcData [10];
USINT    destData[8];

(* Warning! The target memory is smaller than the
   source memory *)

(*Destination address, source addr., length of the data*)
memcpy(adr(destData), adr(srcData), sizeof(destData));
```

When the function is executed, the memory area located at the "srcData" address is copied to the "destData" address. The length of the data (in bytes) was limited by the size of the target memory.

Note:

The **size** of the source data must not be larger than the size of the **target memory**. Otherwise data located behind this memory will be overwritten (e.g. other variables or memory areas).
The "**adr**" function should **always** be used to determine the addresses and "**sizeof**" function to determine the data length.

When uncertain, the data length should always be adjusted to the memory area when copying. This ensures that nothing will be overwritten during copying procedure, because the length of the data to be copied will be limited to this area.

2.5.3 Comparing with "memcmp"

The "memcmp" function can be used to compare memory blocks with each other. The specified lengths of all of the bytes in the specified memory locations are compared with each other.

Example: Comparing memory with memcmp

```
USINT     Data1[10];
USINT     Data2[8];
UINT      status;

(*          Data area 1, data area 2, length of the data*)
status := memcmp(adr(Data1), adr(Data2), sizeof(Data2));

if status = 0 then
     (* Memories have the same content *)
else
     (* Memories have different content *)
end_if
```

When the function is executed, the memory area located behind "Data1" address is compared with the memory area located behind the "Data2" address. The length of the data was limited by the size of the smaller memory (Data2).

When copying or comparing very large memory blocks, it is recommended to optimize these procedures in regard to runtime.
The memory areas can be handled in several separate cycles or divided into smaller units.

2.6 Dynamic Variables / Pointers

In order to make programming more efficient, it is often necessary to work with references to data instead of working directly with the data itself. This is done e.g. when transferring the address of a variable or data area to a function. In this case, only a reference to the location of the data is transferred. If the data is manipulated, it is available to the rest of the program in its new form.

This type of reference is called a pointer or dynamic variable, which points to an area (address) of the application memory (DRAM).

When programming, it is often more efficient and organized to work with dynamic variables.
Proper use of dynamic variables allows you to design software parts in a more intelligent and flexible manner than with static variables.

2.6.1 Functionality

Depending on the programming language, an address in the program code can be accessed differently. However, the principle is always the same.

A **dynamic variable is declared**. Unlike static variables, the compiler does not assign an address to these variables and therefore does not assign a separate memory space either. This is done in the application.
With the key word "**access**", the dynamic variable is **assigned an address**.
The dynamic variable now "**points**" (→ to point → pointer) **to** the assigned **memory location**. The variable is now accessed like a "normal" variable.

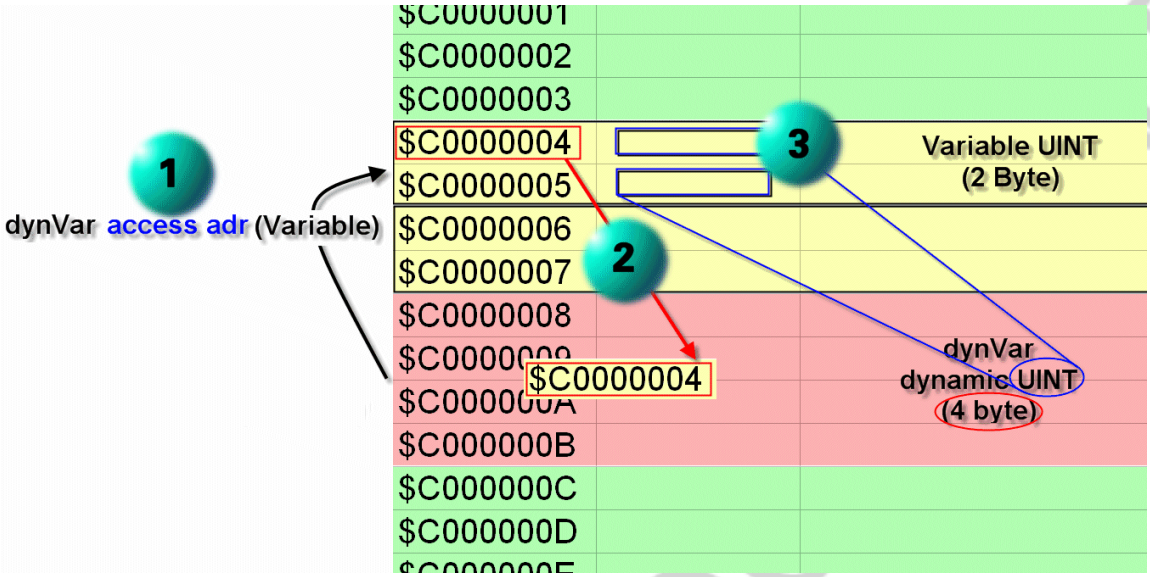


Fig. 17 Functionality of dynamic access

1 The dynamic variable "dynVar" is assigned the address from "variable" via "access".

2 The dynamic variable "dynVar" actually just consists of the address. It has no actual value.

3 The contents of "dynVar" are now also changed when "variable" is changed and vice versa.
The dynamic variable itself is not actually changed but the variable which the dynamic variable is pointing to.

2.6.2 Rules

The following rules apply when working with dynamic variables:

- The compiler **does not assign a separate address** to a dynamic variable. An address must be **assigned** during runtime.
- Once an address has been assigned, it remains valid as long as the system is running. **Cyclic assignment is not necessary**.
- A dynamic variable can be assigned **a new address at any time**.
- **0 is not a valid address**. Check in the application for validity before assigning the memory addresses.
- **Never use fixed memory addresses**. Only addresses which you have obtained using "**adr**" and an optional offset calculation should be used.
- The dynamic variable must be referenced before being accessed for the first time. Dynamic variables are not given an address by the compiler resulting in **unchecked** memory referencing. Writing an unreferenced dynamic variable causes the system to **restart** in **service mode**. An error is then entered in the **logbook**.
- The dynamic variable must have the **same data type** as the data in the source memory area so that the data in the dynamic variable can be correctly interpreted after the dynamic access.

Example: Dynamic access using a auxiliary varible

Method 1:Auxiliary variable for the address

```
UDINT    adrCounter;
USINT    Counter;
USINT*   pCounter;          (* !!Dynamic variable!!   *)

adrCounter := adr(Counter); (* Determines the address *)
pCounter access adrCounter; (* Dynamic access         *)

If pCounter = 17 then  (* Warning!!! dyn. variable must
                          be referenced *)
   (* Program code executed *)
end_if
```

Example: Direct dynamic access

Method 2: Direct access to the address

```
USINT    Counter;
USINT*   pCounter;           (* !!Dynamic variable!! *)

pCounter access adr(Counter) (* Dynamic access       *)

If pCounter = 17 then  (* Warning!! dyn. variable must
                          be referenced *)
   (* Program code executed *)
end_if
```

Example: Checking the address bevor accessing

Method 3: Checking the address before accessing

```
UDINT    adrCounter;
USINT    Counter;
USINT*   pCounter;      (* !!Dynamic variable!! *)

if adrCounter <> 0 then (* Checking for null pointer!  *)
                        (* Address originates e.g. from*)
                        (* another task                 *)
    pCounter access adrCounter; (* Dynamic access *)
    if pCounter = 17 then
        (* ... *)
    end_if
end_if
```

2.7 Memory Allocation

When programming software, just creating arrays or arrays from structures is often not sufficient because these can only be defined with a fixed dimension during the time before being compiled.

It is not possible to create arrays or arrays from structures with more than 4095 array elements. Furthermore, very large arrays will cause the maximum **variable memory for each task** or **the global variable memory** in the project, which is limited to **32 Kbytes on SG3** and **64Kbytes on SG4** systems, to be filled up very quickly.

In this case, it makes sense to create a memory area dimensioned according to the conditions once the system is running.
Use dynamic access of this memory to manage it with the help of offset calculations.

As a result, the software will be somewhat more complex, but also more dynamic and configurable than when using static array variables.

Example: Determining the array dimension

Let's assume that a user data type, which saves the information for a recipe, should be able to save **2000 different recipes**.
Access should occur via an **index** specified by the user.

Name	Type	Scope	Force	Value
Bread	recipe_typ	global		
flour	SINT			120
water	UINT			12
salt	USINT			1
yeast	UDINT			2

```
recipe_typ    Recipes[2000];
UINT          elem;
UINT          index;

elem := sizeof(Recipes) / sizeof(Recipes[0]); (*Amount*)

if (index >= elem - 1) then
    index := elem - 1;      (* Index limitation       *)
end_if

Recipes[index].water := 11; (* Access to the elements *)
```

The size of the entire array determined using the "**sizeof**" function produces 24 Kbytes. Furthermore, the "recipes" variable is **fixed** with 2000 elements and the **size** cannot be adjusted while the system is running.
As an **alternative**, a memory area can now be **allocated** in the desired size. This can be selected as "any" size. The local and global variable memory is not changed as a result of this.

2.7.1 Memory allocation options

There are essentially **two ways** to allocate memory:

- "SYS_LIB" library – "TMP_alloc" function
- "AsMem" library functions

The **TMP_alloc** function can be used to **allocate** a memory area in the application. This function should only be executed in the init subprogram on SG4 systems. Otherwise a cycle time violation could occur.

The functions from the **AsMem** library must be used if the application requires a high speed, **dynamic memory allocation** while the system is running **in cyclic mode**.

Note:

Take note that with cyclic allocation, memory blocks that have been allocated must later be **de-allocated**. Allocated memory blocks are created in the **DRAM**. All allocated memory blocks are automatically de-allocated again after a system restart.

When allocating, make sure that the necessary memory sizes are calculated using the "**sizeof**" function if possible.

Caution:

Newly **allocated memory areas** are generally **NOT initialized**! It is possible to initialize the memory area that was defined using the "**memset**" function.

2.7.2 Working with the "TMP_alloc" function

The "**TMP_alloc**" function is located in the "SYS_LIB" library. Memory blocks requested with the "TMP_alloc" function, can be later de-allocated using the "**TMP_free**" function.

Example: Working with TMP_alloc

Requesting a memory area that has the size of a UINT array with 1000 elements. The memory block is then de-allocated.

```
UINT    uintVar;
UDINT   memlng;
UDINT   memptr;
UINT    status;
UINT*   pData;      (* !! Dynamic variable !! *)

(* init program *)
memlng := 1000 * SIZEOF(uintVar);(* Size calculation *)

(*                Size, pointer to the address variable*)
status := TMP_alloc(memlng, ADR(memptr)); (* Allocation*)

(* Evaluation of the status variable *)
if status = 0 then
    (* Working with the address of the memory area *)
    pData access memptr; (* Dyn. access to the memory *)
else
    (* Error handling, error numbers can be found
       in the help files *)
end_if

(* De-allocating the memory *)
(*                Size   Address of the memory area *)
status := TMP_free(memlng, memptr); (* De-allocate   *)

(* Evaluation of the status variable *)
if ... then
    (* ... *)
end_if
```

Note:

After the system is restarted or the memory area is de-allocated the address determined by "TMP_alloc" must NOT be accessed anymore. The address returned here cannot be used until the "TMP_alloc" function has been executed again.

2.7.3 Working with AsMem

The following section will describe the functionality of the AsMem library in further detail:

A global memory area with the size required for the application is allocated in the **init subprogram** of a task. This memory area can also be called a **memory partition**.
However, it is necessary that the user can use the application to see **how much** dynamic allocatable **memory** is **required** while the system is running.

The individual memory blocks are then **dynamically allocated** from this memory partition while the system is running.

This type of runtime allocation is very **fast** and hardly affects the application runtime.

Example: Working with AsMem

Requesting a 10 KByte memory area.

```
UDINT       ident;
UINT        status;
UDINT       mem;
UINT*       pData;      (* !! Dynamic variable !! *)

(* init program *)
(* Allocation of the memory partition *)
AsMemPartCreate_0(enable:= 1, len:= 10000);
if AsMemPartCreate_0.status = 0 then
    ident:= AsMemPartCreate_0.ident; (* Partition ident *)
end_if

(* cyclic program *)
if cmd_CreateBlock = 1 then
    if AsMemPartCreate_0.status = 0 then
        (* Allocation of a 50 byte memory block
           ident of the memory partition must be specified
           AsMemPartAllocClear initializes the memory block
           with "0" *)
        AsMemPartAllocClear_0(enable:= 1, ident:= ident, len:= 50);
    end_if
    cmd_CreateBlock:= 0;
end_if

(* Working with the allocated memory block *)
if AsMemPartAllocClear.mem <> 0 then
    (* Address of the memory area *)
    mem:= AsMemPartAllocClear_0.mem;
    (* Dyn. access to the memory  *)
    pData access mem;
end_if

if cmd_ReleaseBlock = 1 then
    (* Memory block with the start address "mem" in the partition
       De-allocate with specified ident *)
    AsMemPartFree_0(enable:= 1, ident:= ident, mem:= mem);
    cmd_ReleaseBlock:= 0;
end_if
```

3. DATA STORAGE

3.1 General

As the name suggests, data storage refers to saving data in memory on the controller (mostly ROM and mass memory, but also RAM). This can include data such as machine configuration, recipe data, operating hours, etc.

The user must choose how and on which target memory the data should be saved. In most cases, nonvolatile memory is used for this. The following possibilities are available for long-term data storage on B&R controllers:

- B&R data objects
- Files
- Variables (remanent and permanent)

3.1.1 B&R data objects

Data objects consist of a header, the actual data and a checksum.

Fig. 18 Structure of a data object

The checksum on all B&R objects (including tasks and system objects) is monitored cyclically while the system is running to detect and react to errors such as unauthorized access with pointers. The monitoring is carried out in the idle time. 512 bytes per system tick are checked. Therefore, it can take up to a few minutes (depending on the size of the application) before the checksum monitor responds in the case of an error.

Note:

Checksum monitoring can be switched off for data objects created in RAM during runtime if they are to be written accurately with pointers. In this case, a data object is no different than allocated memory.

Data objects are the safest way to save data because a backup copy is automatically made as part of the application. Therefore, if a data object is destroyed during a write procedure (e.g. in the event of a power failure), a backup copy of the data object is still available and is automatically recovered when the controller is restarted (see the "Data Consistency" section).

3.1.2 Files

On SG4 systems it is possible to store files in mass memory on the controller. Mass memory includes hard drives (HDD), floppy disk drives (FDD), Compact Flash cards or USB storage media. The data is organized as on a PC and saved on logical drives (also in folders if desired).

Files can also be edited using a PC and transferred back to the controller when needed. This is a big advantage when e.g. recipes need to be created once and then transferred to other identical machines.

The following options are available for doing this:

- Removing the Compact Flash and connecting to a PC using a suitable adapter.
- Copying the files from the Compact Flash card to USB storage using the "FileIO" library.
- Access to the Compact Flash card via Ethernet FTP.
- The PLC copies dthe Data to a FTP server.

Using files makes it possible to easily manipulate and exchange data between controllers and PCs.

3.1.3 Variables

Remanent and permanent variables offer the further possibility of protecting data from loss. Files are stored in SRAM (battery-backed part of RAM), thereby being protected from power failures.

The status of the backup battery should be monitored when using remanent or permanent variables to backup data. The "HWGetBatteryInfo" function block from the "AsHW" library can be used to do this.

Note:

RAM is only recommend for data backup under certain conditions because it is a volatile type of memory. Therefore, the rest of this section will deal with the possibility of using ROM to store data in data objects and files.

3.2 Data objects

3.2.1 Creating data objects with Automation Studio

Data objects can be created in Automation Studio while setting up the project or using the "DataObj" library while the system is running. Special attention should be given to byte alignment when creating data objects in Automation Studio (filler bytes, null termination of strings) so that the data ends up in the correct elements of the structure when transferred.

```
;-------------------------------------------------------------
;------------------------ RECIPE DATA ------------------------
;-------------------------------------------------------------
;
;       +---------------------------------------- Description
;       |                                          [STRING(12)]
;       |        +--------------------------------- Bread type
;       |        |                                  [USINT]
;       |        |       +------------------------- Water
;       |        |       |                          [UINT]
;       |        |       |       +----------------- Flour
;       |        |       |       |                  [UDINT]
;       |        |       |       |       +--------- Yeast
;       |        |       |       |       |          [USINT]
;       |        |       |       |       |    +---- Filler Byte
;       |        |       |       |       |    |     [USINT]
;       |        |       |       |       |    |     +- Salt
;       |        |       |       |       |    |     |  [UINT]
;       |        |       |       |       |    |     |
"White Bread ", 001, 00200, 00035000, 010, 000, 00080;
"Brown Bread ", 002, 00300, 00078000, 025, 000, 00110;
"Grain Bread ", 003, 00350, 00125000, 030, 000, 00175;
```

Fig. 19 Data object in Automation Studio

3.2.2 Formatting of the Data

Name	Description
Strings	Strings can be written between quotation marks (**"White Bread"**) or apostrophes (**'White Bread'**). The null termination for the string is automatically inserted when using quotations. The string is stored in the data object without null termination when using apostrophes.
Numeric decimal values	Numeric values are separated with a comma (**001, 00200**) or a line break. Negative / positive values can we written the following way **(-001, +100**).
Hexadecimal values	Hexadecimal values can be signed with the $ - character **($FFF, $FACE, $01FB**).
Binary Values	Binary values can be signed with % - character **(%01010101, %0010010010010000**).
Floating point numbers	Floating point numbers (REAL) are defined with a period as decimal point (**12.3456 or 8.0**). Orders of magnitude are specified with an "e" preceding the exponent (**2.34e5 or 5.43e-21**).
Comments	Comments are started using a semicolon (;) and apply to the rest of the line.
Spaces	Spaces between numeric values are not taken into consideration. Of course, spaces keep there bit value in strings.

It should be possible to read each line in this data object individually with a structure and an offset. Make sure that the lengths and offsets of the variables in the structure exactly match the data in the data object. When reading the data, the structure can be used as a mask.

Name	Type	Scope	Force	Value
RecipeStructure	Recipe_type	local		
Description	STRING[12]			'White Bread '
Type	USINT			1
Water	UINT			200
Flour	UDINT			35000
Yeast	USINT			10
FillerByte	SINT			
Salt	UINT			80

Fig. 20 Structure for reading the data object

A filler byte is automatically inserted after the "Yeast" element (alignment), which must also be taken into consideration in the data object or the structure. Otherwise, the structure can be designed so that it does not contain any filler bytes.

Note:

Numeric values:
The data type being used for numeric values is automatically defined by the number of digits or the by the actual value.
1 byte: Max. 3 digits and value range for SINT or USINT.
2 bytes: Max. 5 digits and value range for INT or UINT.
4 bytes: Max. 10 digits and value range for DINT or UDINT.
(Floating decimal numbers (REAL) also occupy 4 bytes.)

To read a 1 byte value (e.g. 200) with a UINT (2 bytes) variable, it must be entered in the data object with at least four digits (0200).

Strings:
When working with strings it is important that each entry in the data object is exactly as long as the length of the string that is being read (null termination in the variable and in the data object, filler bytes behind string variables).

3.2.3 Creating data objects during runtime

The "**DataObj**" library provides function blocks for creating, managing and editing data objects while the system is running. This library can be used to access existing data objects, to read from or write to data objects; Data objects can be created, deleted copied and moved to another memory.

Note:

Data objects created during runtime can be uploaded to the project from the controller if necessary. However, the contents cannot be displayed by Automation Studio because it is not possible to recompile the data.

3.2.4 Target memory

When working with data objects, you must decide in which target memory to save the data. The desired memory type can be specified in Automation Studio or as an input parameter in the "DatObjCreate" function block. The following table can help you to make this decision.

	Data retained during warm restart	Data retained during cold restart
DRAM		
UserRAM (SRAM)	☑	
UserROM, SystemROM (FLASH)	☑	☑

3.2.5 Working with data objects

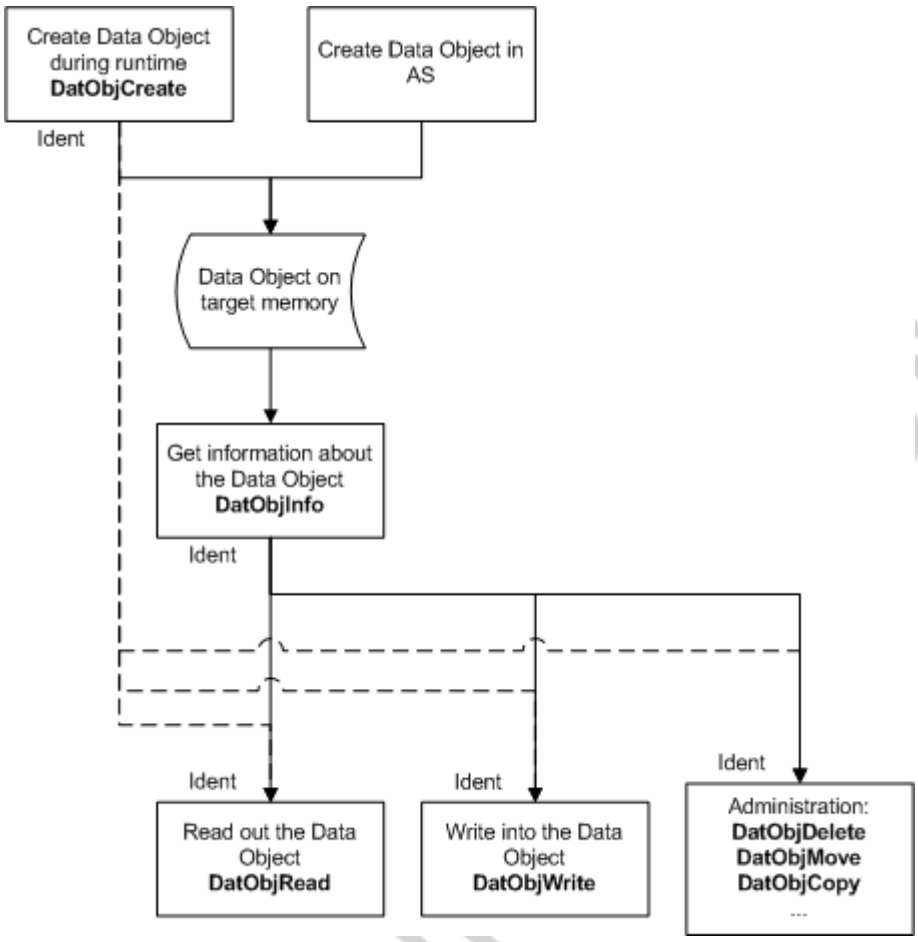


Fig. 21 Using the DataObj library

Creating a data object and reading the information:
The **"DatObjCreate"** function block from the "DataObj" library is used to create a data object while the system is running. The necessary information for further editing can be obtained from an existing data object using the **"DatObjInfo"** function block.

Reading and editing:
The data object can continue to be edited after receiving the ident.
"DatObjRead" is used to read data from the data object, while **"DatObjWrite"** is used to write data to the data object.

Managing data objects:
Additional function blocks are available for managing data objects (deleting, copying, changing the date, moving to another memory).

A detailed description about how to use the function blocks can be found in the Automation Studio online help under **B&R Software World:Automation Studio:Libraries:DataObj**.

Note:

You must check the status output of all function blocks in the **DataObj** library to see if the function block is finished or has been correctly executed (when status = 0). Otherwise, the function block must be executed again in a later cycle or the error must be evaluated.

Dynamic access:
The memory area of a data object can also be accessed directly using dynamic variables (pointers). The "DatObjCreate" and "DatObjInfo" function blocks provide the start address for the data via the "pDatObjMem" output.

True **read access** with dynamic variables does not require any other special considerations.

However, a few things must be considered regarding **write access** with dynamic variables. As mentioned earlier, the **checksum** of data objects is monitored cyclically. That means that data objects can only be write accessed when checksum monitoring is switched off. This is only possible for data objects in the DRAM or UserRAM if they are created while the system is running.

Note:

We recommend only using the corresponding library functions to edit data objects. If dynamic variables must be used to manipulate data areas created during runtime, it makes more sense to allocate a memory using the "TMP_alloc" function and (if necessary) to save in a data object using the library functions after editing.

Task: Reading the content of a Data Object during runtime

Create a data object and data type in Automation Studio as shown here.

```
;-------------------------------------------------------------
;------------------------ RECIPE DATA ------------------------
;-------------------------------------------------------------
;
;        +---------------------------------------- Description
;        |                                          [STRING(12)]
;        |         +------------------------------- Bread type
;        |         |                                [USINT]
;        |         |       +----------------------- Water
;        |         |       |                        [UINT]
;        |         |       |        +-------------- Flour
;        |         |       |        |               [UDINT]
;        |         |       |        |       +------ Yeast
;        |         |       |        |       |       [USINT]
;        |         |       |        |       |    +- Filler Byte
;        |         |       |        |       |    |  [USINT]
;        |         |       |        |       |    |     +- Salt
;        |         |       |        |       |    |     |  [UINT]
;        |         |       |        |       |    |     |
"White Bread ", 001, 00200, 00035000, 010, 000, 00080;
"Brown Bread ", 002, 00300, 00078000, 025, 000, 00110;
"Grain Bread ", 003, 00350, 00125000, 030, 000, 00175;
```

Name	Type	Scope	Force	Value
RecipeStructure	Recipe_type	local		
Description	STRING[12]			'White Bread'
Type	USINT			1
Water	UINT			200
Flour	UDINT			35000
Yeast	USINT			10
FillerByte	SINT			
Salt	UINT			80

Write a task for reading the 3 recipes individually! Use the **DatObjRead** function block to do this!

Also read the data with a **dynamic variable structure**!

Use the **DatObjWrite** function block to change different recipe parameters!

Check the changes with the **dynamic variable structure**!

Approach:

The "**DatObjInfo**" function block provides the necessary information for the other function blocks and dynamic access.

The offset of the second and third recipe and the length of the data block being read are easily determined using the "**SIZEOF**" function. Now the recipe's index just has to be defined (starting at 0).

3.3 Files

B&R SG4 controllers are equipped with mass memory (Compact Flash, hard drive, USB memory). They offer the possibility to store and to organize files on these memory media, as is common in the PC world. The **"FileIO"** library offers function blocks for using the extensive file system functions on the mass memory.

Files can generally be used for the same purpose and in a similar manner to data objects, however, there are differences in the format and the management by the Automation Runtime and the user.

	Files	Data objects
Memory	Mass Memory	UserROM, SystemROM, UserRAM, DRAM
Access	Function blocks and dynamic variables	Function blocks and dynamic variables
Controller system	SG4	SG3, SG4
Monitoring, Checksum	No	Yes
Library	FileIO	DataObj
Can be transferred to PC	FTP, Compact Flash, USB	No
Can be edited on PC	Any program	Automation Studio

As described earlier, files make it possible to transfer data from the control system to a PC, where they can be saved or edited, or passed onto another control system.

3.3.1 File devices

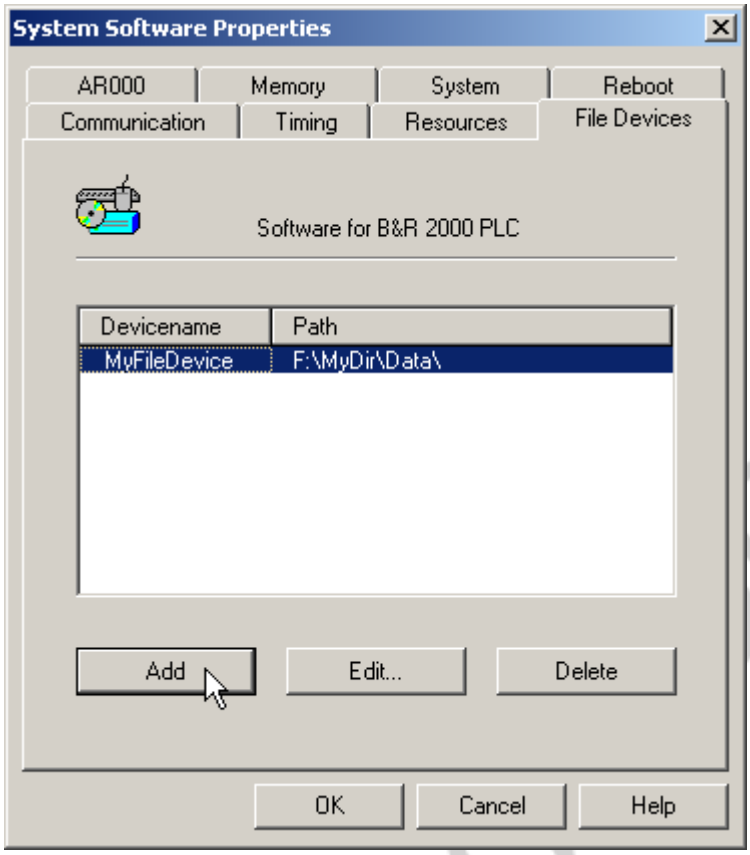


Fig. 22 File device

A connection between the file system and Automation Runtime or the application must first be established in order to use the file system functions. Therefore, files devices must be setup in the CPU settings.

The device name is used in the "FileIO" library as a synonym for the highest directory level. The operating system then assumes access to the files.

A device can also contain sub-directories that can be directly accessed using the function blocks.

Note:

Detailed information about file devices can be found in the Automation Studio online help under **Automation Software:Automation Studio:Projects Software configuration:SG4 CPU configuration:File devices**.

3.3.2 Working with files

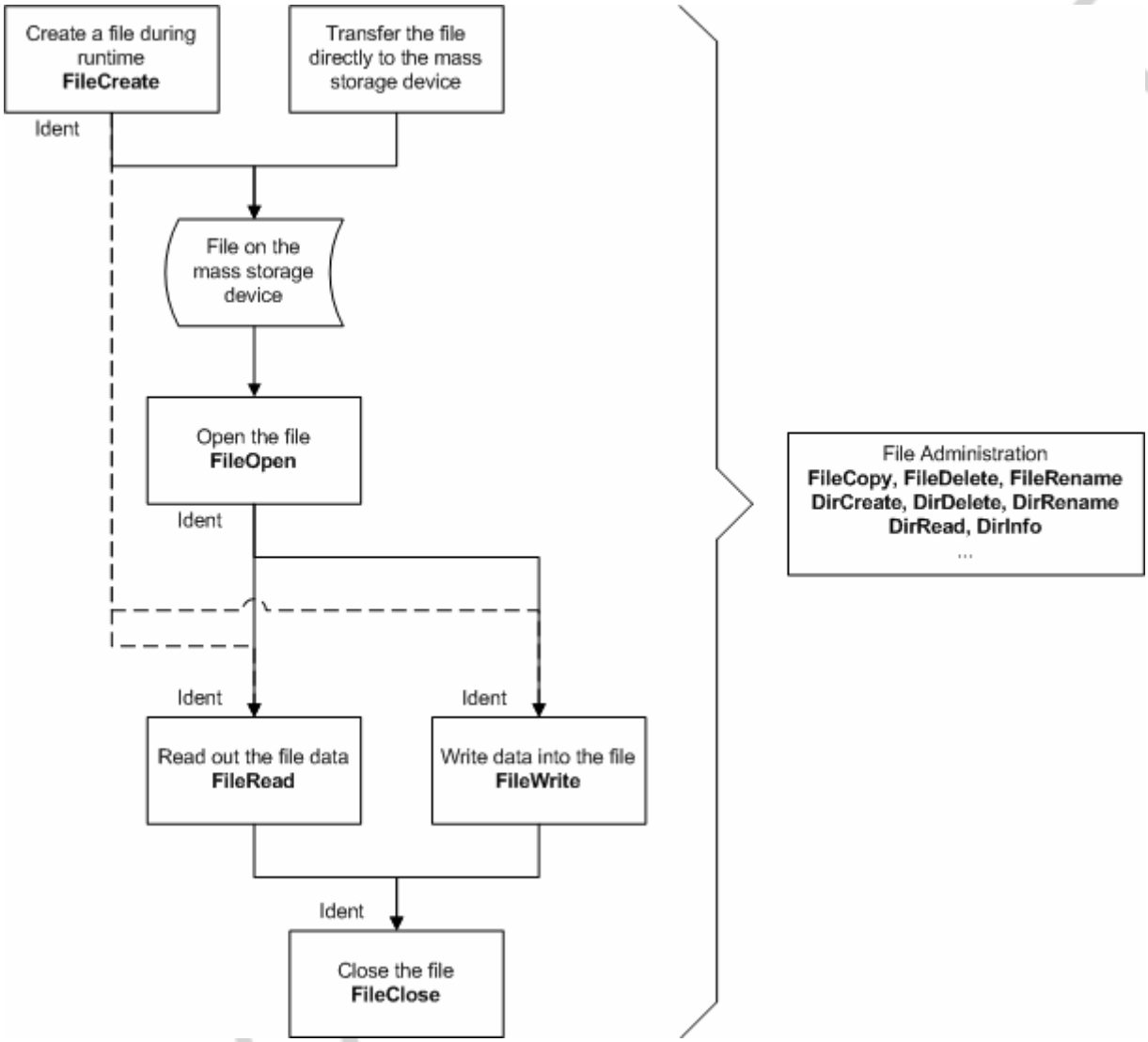


Fig. 23 Using the FileIO library

Creating a file:
The **"FileCreate"** function block from the "FileIO" library is used to create a file while the system is running. At this point, the file is given a name and the ident received provides the basis for further editing.

Opening an existing file:
A file can be opened from the mass memory using the **"FileOpen"** function block. The file can be further edited when the ident is received. "FileOpen" is not necessary if the file was just created using "FileCreate".

Reading and editing:
The ident makes it possible to read data from files using **"FileRead"** and to write data to files using **"FileWrite"**.

Closing a file:
A file opened with "FileOpen" or created with "FileCreate" can be closed using **"FileClose"**.

Caution:

After editing a file, the file should be closed to make the system resources required by "FileOpen" or "FileCreate" available again.
This helps you gain a clearer overview when using multiple files.

Managing files:
Additional function blocks are available for managing files (deleting, copying, renaming), organizing directories (creating, deleting renaming a directory) and reading the contents of a directory.

A detailed description about how to use the function blocks can be found in the Automation Studio online help under **Automation Software:Automation Studio:Libraries:FileIO**.

Note:

You must check the status output of all function blocks in the **"FileIO"** library to see if the function block is finished or has been correctly executed (when status = 0). Otherwise, the function block must be executed again in a later cycle and/or the error must be evaluated.

3.3.3 File formats

The format for storing data in the files must be determined if the files are to be edited externally (e.g. on Windows, Linux or Unix platforms). It is possible to store data directly like it is located in the variable structure on the controller **(binary)**, or to convert it to text first **(ASCII)**. Text formats may be excluded according to the editor being used (CSV, XML, etc.).

The following functions are available in the "**AsString**" library for converting numeric values to text and vice versa:

- itoa: Converts an integer value (max. DINT) to an ASCII string.
- atoi: Converts an ASCII string to an integer value (DINT) .
- ftoa: Converts a floating point number (REAL) to an ASCII string.
- atof: Converts an ASCII string to a floating point number (REAL).

Example: Binary data in a file

In the following example, the contents of the structure, "FileData" will be saved three times in the file.

Name	Type	Scope	Force	Value
FileData	DataType_typ	local		
Text	STRING[32]			'Second entry in file:'
Numeric	INT[4]			
Numeric[0]	INT			643
Numeric[1]	INT			-7452
Numeric[2]	INT			1234
Numeric[3]	INT			7653

Fig. 25 Data in the structure

Without being converted, the data is displayed in a text editor as follows (the line breaks were entered afterwards):

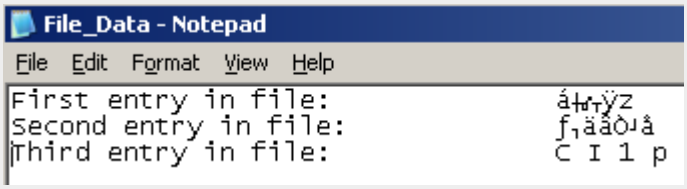


Fig. 25 Binary data in the file

A "CSV file" (CSV → Comma Separated Values) is created when the numeric variable values are first converted to text (e.g. using the "itoa" function) and a semicolon is inserted after each value for clear separation of the values. This format is especially well-suited for editing in a table editor such as Microsoft Excel.

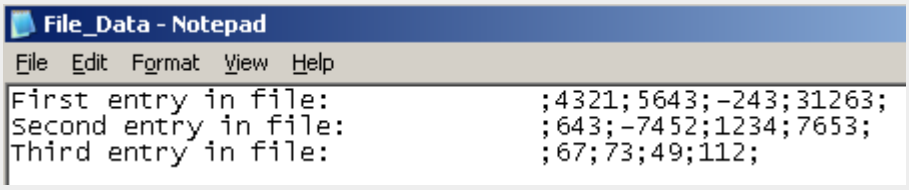


Fig. 26 Data converted to ASCII in a CSV file

When reading the data however, each individual value must be read from the file, converted back to the numeric value (e.g. using the "atoi" function) and copied to the correct variable in the structure.

Note:

In both cases, the data is read in the opposite order as the data was written. However, the process for text formats is somewhat more complicated. As you can see, the data requires more memory space when it is in text format. A numeric 2 byte value can use up to 6 places (5 digits plus sign = 6 bytes). Serparators and line breaks must also be taken into consideration.

Task: Create, Read, Write a file with the FileIO Library

Configure a **file device** in the CPU settings!

Create a data type in Automation Studio according to the example shown above!

Write a task for creating a file (**FileCreate**) / opening an existing file (**FileOpen**) and closing an open file (**FileClose**)!

Use a structure from the created data type to write data to the file using the "**FileWrite**" function block!

Transfer the file to the PC and edit the file using a **text editor**!

Approach:

The offsets and the lengths of the data blocks being read are easily determined using the "**SIZEOF**" function. Now the data block's index just has to be defined (starting at 0).

Caution:

When editing the file in an external editor, make sure when saving that:

- the offsets of the individual values match the structure being read.
- the editor being used does not automatically replace any characters (e.g. Notepad: binary 0 (Null) replaced with spaces (hex 20, dec 32)). This would result in missing null terminations in the string when reading.

3.4 Data consistency

3.4.1 General Information

A file system is used on B&R controllers for organizing all data such as application program, operating system, configuration settings and user data.

On computer systems a file system is used for saving, managing and locating files. It is possible to use names for organizing and accessing files on mass memory devices such as hard drives or Compact Flash cards. File names can also be stored in special files in directories. This makes it possible to call all files using unique names (file name including path).

Each task, each data object, etc. is saved as a file in non-volatile memory.

Considerations must be made regarding protection against failure and limitation of damage in the event or an error because parts of the application can be edited and saved while the system is running (e.g. data objects).

It is always possible that a power failure could occur while writing a data object during a write access to the effect file. If this occurs, the file is destroyed and the data is lost.

This problem does not exist for the application data because the controller always contains a current backup copy. This means that the last saved version is automatically restored when an error is detected.

This is not the case for user data (files). In the case of user data, the data is lost when a power failure occurs during a write procedure.

Note:

To guarantee supply voltage, B&R offers **uninterruptible power supplies** (UPS), which detect a power failure and notify the controller. This makes it possible for the application program to be secured before the voltage is lost.

3.4.2 Accessing mass memory

Access to peripheral devices such as mass memory is handled by the operating system with very low priority.

These devices are controlled using function blocks that can return the value "busy" (= 65535) to the status output. That means that the function block must be called again in the next cycle until the value 0 or an error number is returned to the status output.

If the status 0 is returned, it means that the operating system has accepted the task. However, the action does not have to be actually executed at that exact moment. The action will be executed as soon as sufficient system resources are available.

As a result, on CPUs with a high load the function block could return the status 0, but the edited file is not updated if the supply voltage is interrupted in the meantime.

Caution:

The fact that Flash memory can only be written a limited number of times must be taken into consideration when using Flash (e.g. Compact Flash) as mass memory. Therefore, cyclic write access from the application must be kept to the necessary minimum.
The actual number of write accesses can be found in the Flash card's documentation.

Fig. 27 Compact Flash

3.4.3 Directory structure on the Compact Flash

On SG4 systems, all of the data that must be stored long-term is saved on Compact Flash (or another mass memory) in a fixed directory structure.

One partition system:
If a Compact Flash contains one or two partition(s), then the entire control software is stored on the first partition (C:\). The second partition (D:\) can be used as a user partition to store e.g. files created in the application.

The following directory structure is created when downloading the operating system to the first partition (for the CPU C:\) on the Compact Flash (view in Windows Explorer):

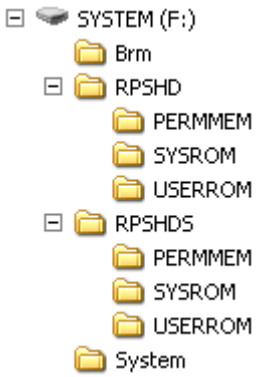


Fig. 28 Directory structure on the Compact Flash

- Brm:
  Configuration files for the operating system.
- RPSHD:
  The application is saved in this directory and loaded completely to the DRAM from here during startup.
- RPSHDS:
  The backup copy of the application is saved here. The application is restored from this directory if an error is detected in RPSHD during startup (e.g. incorrect checksum).
- System:
  Firmware and drive files for built-in interfaces and optional interface cards are stored here.

Three partition system:
If the Compact Flash is divided into three or more partitions and if the third partition (E:\) is the same size or larger than the second (D:\), then the entire control software is placed on the first three partitions. The remaining partitions can be used as user partitions. The operating system (files in the root directory, Brm\ and System\) is stored on the first partition (C:\), the application (RPSHD\) on the second (D:\) and the backup copy of the application (RPSHDS\) on the third (E:\).

The user data can be organized on a fourth partition (F:\) separate from the control software.

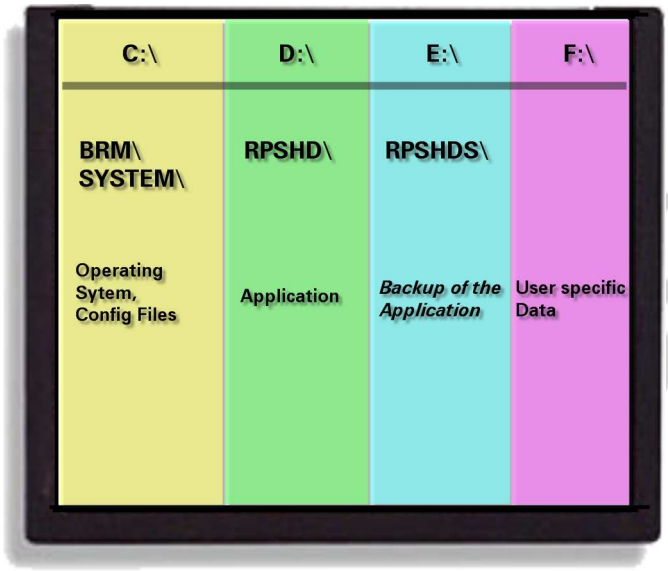


Fig. 29 Three partition system with user partition

The three partition system offers the highest level of security by clearly separating the operating system, application, backup copy of the application and user data on four different partitions. As a result, any potential sources of error in the file system (FAT) are limited to one partition. In the event of an error, the control software can always be restored from another partition.

Note:

The compact flash can be created with the help of the PVI Transfer Tool. It provides the choice of the partition sytem, selection of the individual partition sizes as well as the content of the user partition. Further information you can find in the Automation Studio Online help and the PVI Transfer help.

4. SUMMARY

Solid knowledge of memory management, data storage and their respective features can help to prevent programming errors right from the start. This eases the amount of work and provides more certainty when planning data storage on the controller.

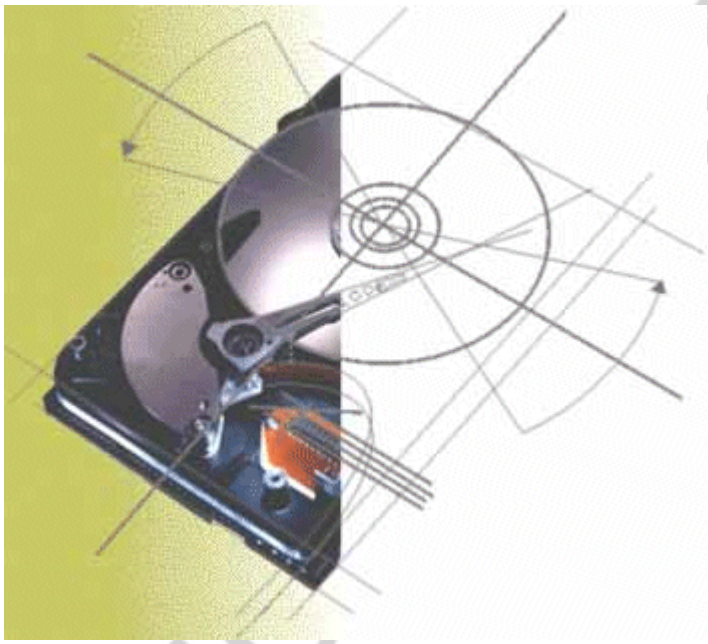


Fig. 30 The parts in a hard drive

Participants now have an overview of the structures, arrays and dynamic variables. Sources of error in an application program can be limited by working carefully with the memory. A suitable method of data storage in the form of data objects or files offers further advantages for your application.

Overview of training modules

TM200 – B&R Company Presentation **
TM201 – B&R Product Spectrum **
TM210 – The Basics of Automation Studio
TM211 – Automation Studio Online Communication
TM212 – Automation Target **
TM213 – Automation Runtime
TM220 – The Service Technician on the Job
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Generation
TM240 – Ladder Diagram (LAD)
TM241 – Function Block Diagram (FBD)
TM246 – Structured Text (ST)
TM247 – Automation Basic (AB)
TM248 – ANSI C
TM250 – Memory Management and Data Storage
TM260 – Automation Studio Libraries I
TM261 – Closed Loop Control with LOOPCONR

TM400 – The Basics of Motion Control
TM410 – The Basics of ASiM
TM440 – ASiM Basic Functions
TM441 – ASiM Multi-Axis Functions
TM445 – ACOPOS ACP10 Software
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Starting up Motors

TM500 – The Basics of Integrated Safety Technology
TM510 – ASiST SafeDESIGNER

TM600 – The Basics of Visualization
TM610 – The Basics of ASiV
TM630 – Visualization Programming Guide
TM640 – ASiV Alarm System
TM650 – ASiV Internationalization
TM660 – ASiV Remote
TM670 – ASiV Advanced

TM700 – Automation Net PVI
TM710 – PVI Communication
TM711 – PVI DLL Programming
TM712 – PVIServices
TM730 – PVI OPC

TM800 – APROL System Concept
TM810 – APROL Setup, Configuration and Recovery
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM840 – APROL Parameter Management and Recipes
TM850 – APROL Controller Configuration and INA
TM860 – APROL Library Engineering
TM865 – APROL Library Guide Book
TM870 – APROL Python Programming
TM890 – The Basics of LINUX

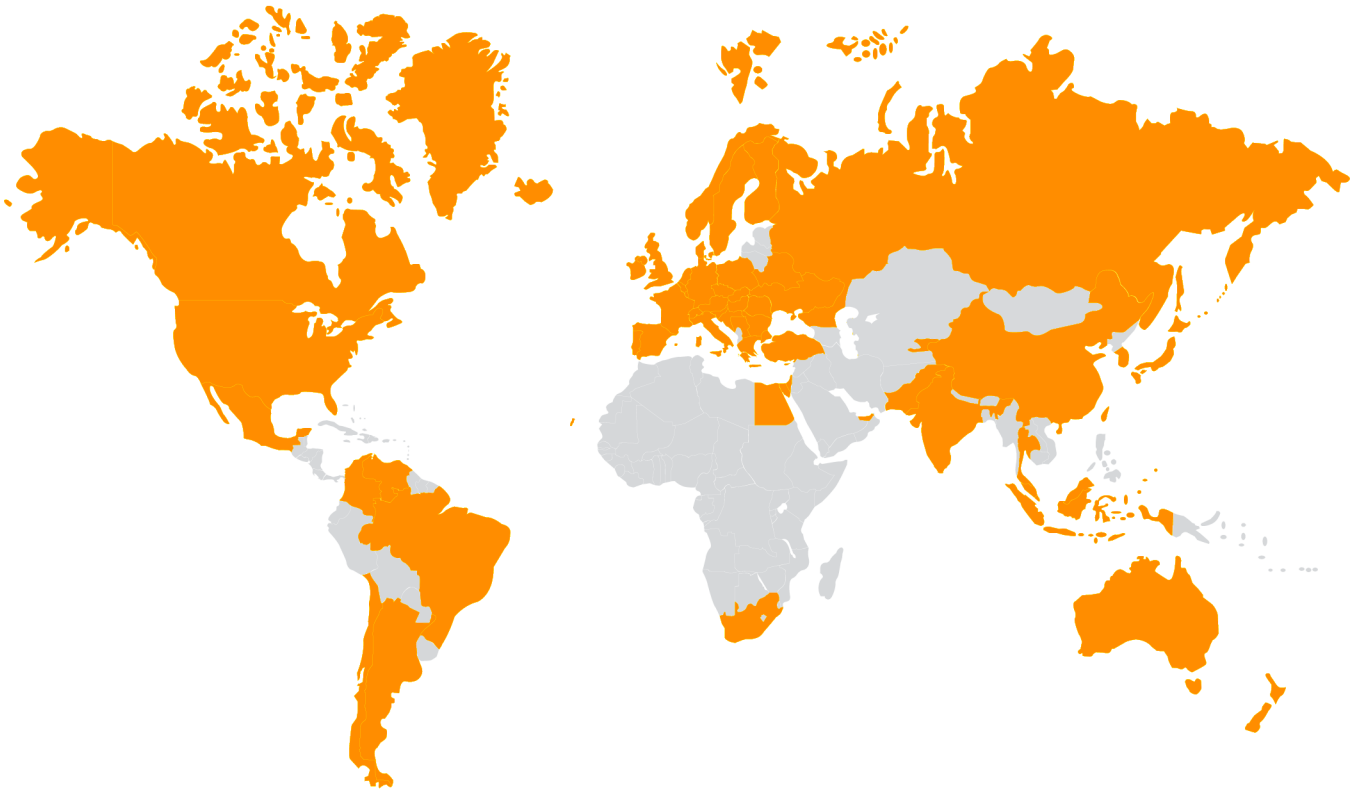
**) see Product Catalog

CORPORATE HEADQUARTERS

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

TM250TRE.00-ENG 0907

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



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