Automation Runtime

TM213

Perfection in Automation

www.br-automation.com

Requirements

Training modules:	TM210 – The Basics of Automation Studio
	TM211 – Automation Studio Online Communication
Software:	Automation Studio 2.6 / 3.0
	Automation Runtime 2.90
Hardware:	None

Table of contents

1. INTRODUCTION

On all PC and controller systems, the operating system serves as the interface between the user and the hardware, thus making it the foundation that must be present before any system can be used.

It provides the basic functionalities and is responsible for the management of resources. Some of its most important tasks include scheduling and allocating memory and time slots.

Demands on operating systems in the area of control include a modular structure and the ability to quickly execute the application repeatedly within a precise timeframe. For the user, this brings the highest production numbers, quality, and accuracy.

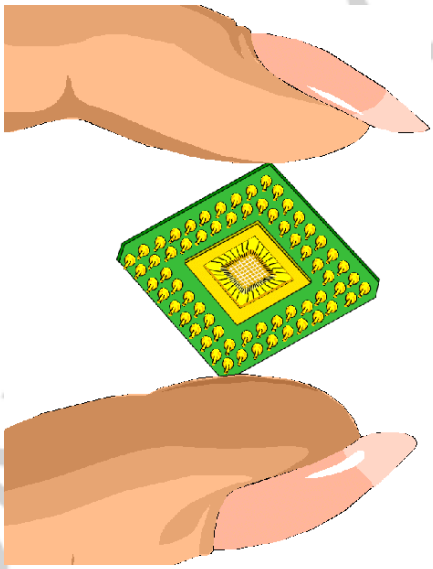


Fig. 1 Microchip

This training module is meant to provide a general overview of operating systems and their characteristics.
By giving an exact description of the operating system known as Automation Runtime and explaining how to use the different interaction and configuration options, the user will be able to make adjustments to his application.

1.1 Objectives

You will learn that there is a close relationship between accuracy, the number of machine cycles, and the quantity/quality of a product.

You will also become acquainted with Automation Runtime and gain a better understanding of how it works.

You will learn about the configuration options possible in Automation Runtime.

And you will learn how to update Automation Runtime.

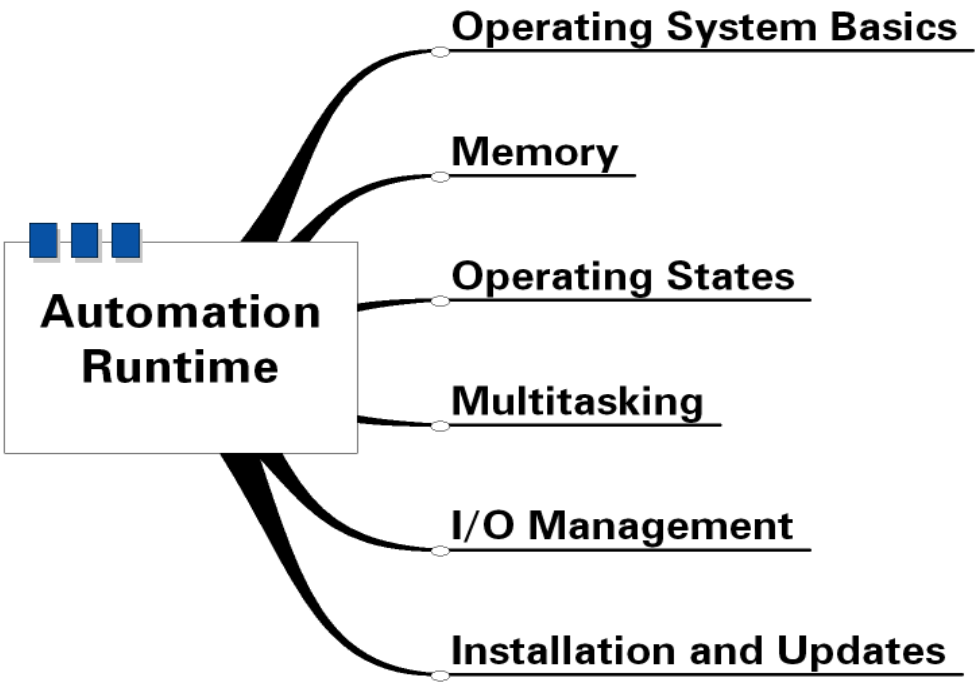


Fig. 2 Overview

2. OPERATING SYSTEM BASICS

Every computer requires an operating system before it can be used. This provides the user with functionalities that make it easier to use the hardware. The more functions that the operating system handles, the faster and easier it is for the user to create software. Functions that are handled by the operating system do not have to be programmed by the user. This shortens the amount of time needed to create a program and reduces the number of errors.

That's why operating systems are now appearing in ever smaller electronic devices.

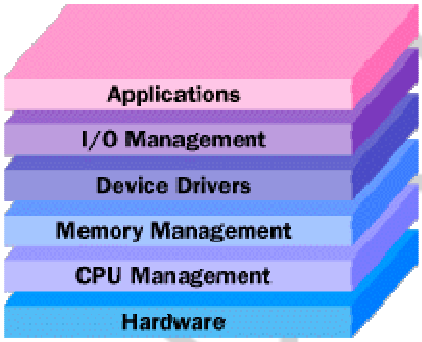


Fig. 3 Operating system architecture

An operating system essentially has two main tasks. On the one hand, it manages a system's hardware and software resources. On the other hand, it provides the user and his application with a stable and uniform way of accessing hardware without having to know the details.

There are different types of operating systems, which are differentiated according to their characteristics and how they work.

A **real-time multitasking** operating system runs on B&R Automation Targets. This operating system is also called **Automation Runtime** (AR).

All program modules, also called tasks, are assigned to different task classes. These task classes determine the time frame in which the tasks will be processed. The time frame is always the same and is separate from the execution time of the individual tasks. This is the meaning of "real-time". The operating system is thus responsible for timing stability on the controller system.

2.1 Demands in the automation industry

In the harsh field of automation, there are several demands placed on a controller's operating system that need to be met.

These demands include the following:

- Modularity in the form of libraries
- Structuring possibilities in the application software
- Performance: The operating system should require few resources so that the majority can be used for the application.
- Low jitter: Timing stability for the execution of individual task classes (time slots where the programs are executed)
- Short cycle times
- Configuration options for the timing behavior
- Possibility to operate peripherals (interfaces, file system, etc.)
- Diagnostic options
- Uniform runtime system even on different platforms

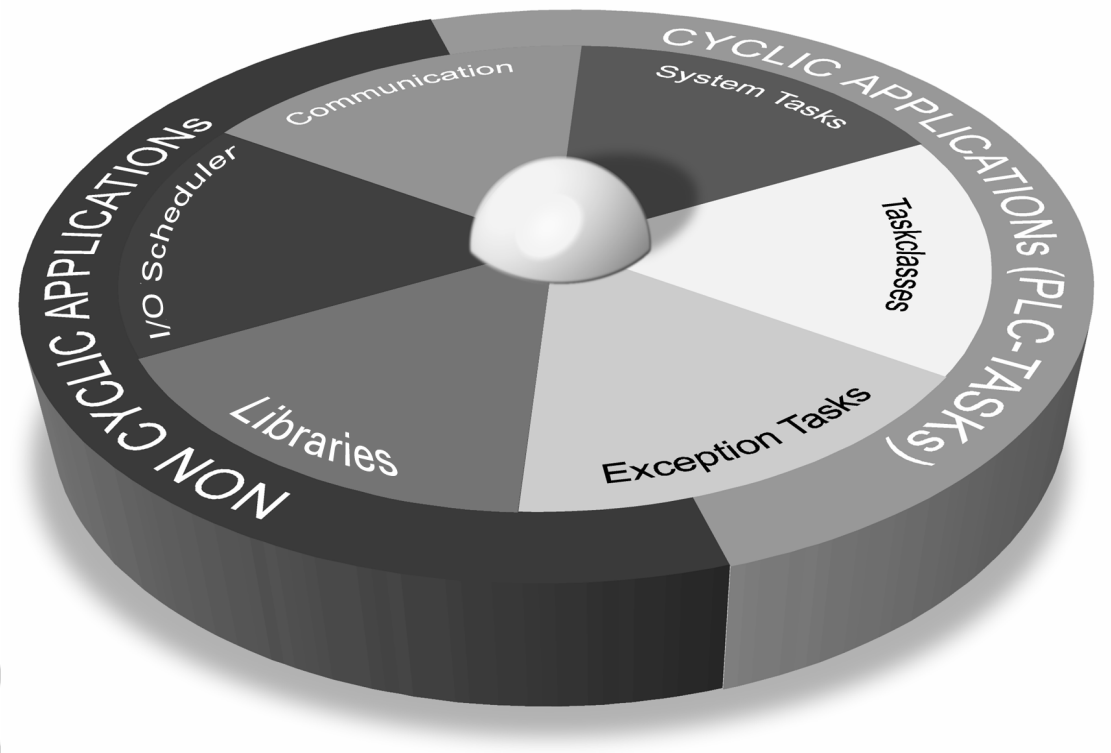


Fig. 4 Automation Runtime

Because B&R Automation Runtime is able to meet all of these demands, it remains the optimal solution.

2.1.1 Advantages of Automation Runtime

- Real-time operating system
- Modular software architecture
- Integrated system monitoring
- Platform-independent
- Debugging tools

2.1.2 Using different platforms

A platform basically means the processor architecture or and processor type that a particular system is using.
The following list shows how these platforms can be different:

- Processor types
- Data management
- Task class types
- Boot behavior
- Watchdog
- Arithmetic operations
- Graphics capabilities
- Network possibilities
- Address management
- ...

B&R uses two different platforms: SG3 (System Generation) and SG4 targets.

For the **user** however, there is **no considerable difference** between the two platforms due to the uniformity of the programming system.

Note:

Details about these differences can be found in the online help under **Autmation Software:Automation Runtime:General**.

3. MEMORY

Each Automation Target requires a memory area. This is where various software elements are stored:

- Automation Runtime
- Controller program
- Variable values
- Tables
- Recipes
- Data objects
- ...

3.1 Storage media

Different storage media can be used on the automation targets. They can be roughly divided into RAM and ROM. Sections of each of these memory areas are available to the user while others are exclusively reserved for the operating system and cannot be used by the user.

3.1.1 RAM

RAM is a high-speed read/write memory which mostly contains the data for executing programs. Data stored here is only kept as long as the memory is provided with power. It's divided into the following:

- DRAM
  DRAM is RAM memory that can be used to store software objects, variable values, tables, etc. It is distinguished by its high-speed access. This type of memory is not battery-backed, which means that all data is lost in the event of a power failure.
- SRAM
  Also called static RAM. Unlike DRAM, SRAM is battery-backed. Data is maintained as long as the buffer is working properly.

3.1.2 ROM

ROM memory is slow and is used for long-term storage of data. The data is also maintained even in the absence of power supply. Flash (Compact Flash) is a typical type of ROM memory. The operating system and the application are stored here.

3.2 Division of memory

In Automation Studio, RAM and ROM are divided further into logical groups. Each contains a system area and a user area.

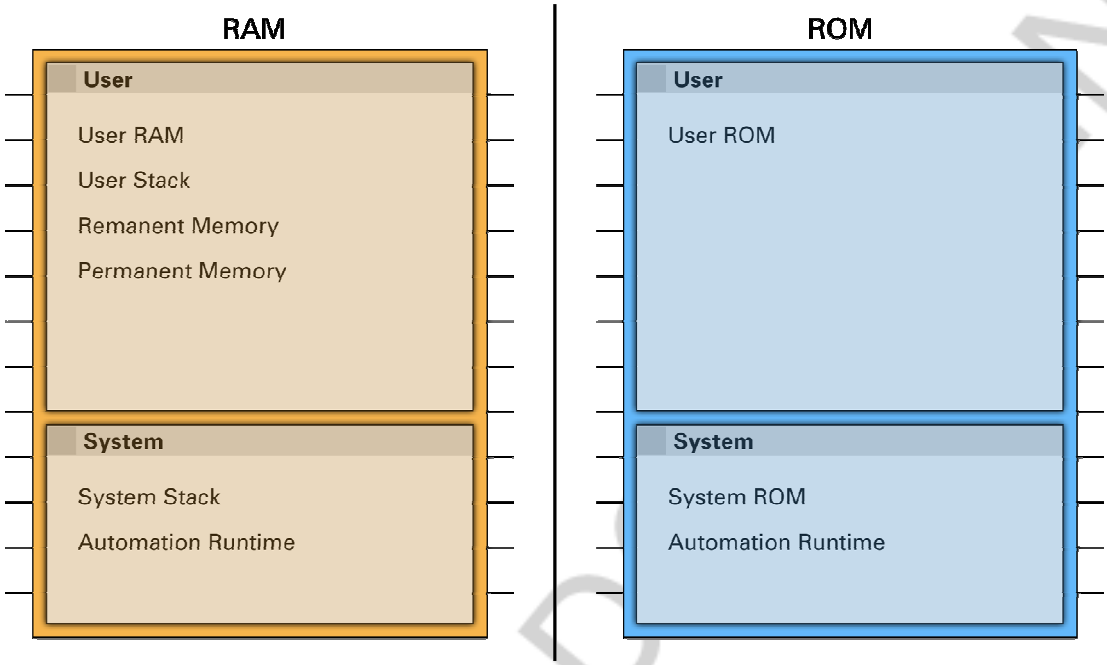


Fig. 5 Memory management

User areas are logical sections of memory where user tasks (ROM); remanent, permanent, and volatile variables (RAM); and data objects (RAM and ROM) can be stored.

System areas are logical sections of memory used to store the operating system (ROM) and system-relevant, permanent, and temporary data (RAM). These sections are intended solely for the operating system.

3.3 Variables and constants

Variables and constants are symbolic elements used in programming whose structure and size are determined by a data type. They are assigned a physical place in memory by the compiler. There are different memory areas that behave differently when a system is booted and at runtime.

3.3.1 Global and local variables

Global variables can be accessed from any task in a control application. A global variable is guaranteed to be recognized by all tasks and to have the same memory address because they are stored in an extra memory area.

Local variables are only recognized in their own task. Other tasks cannot access local variables being used in another task, which is why there are global variables.

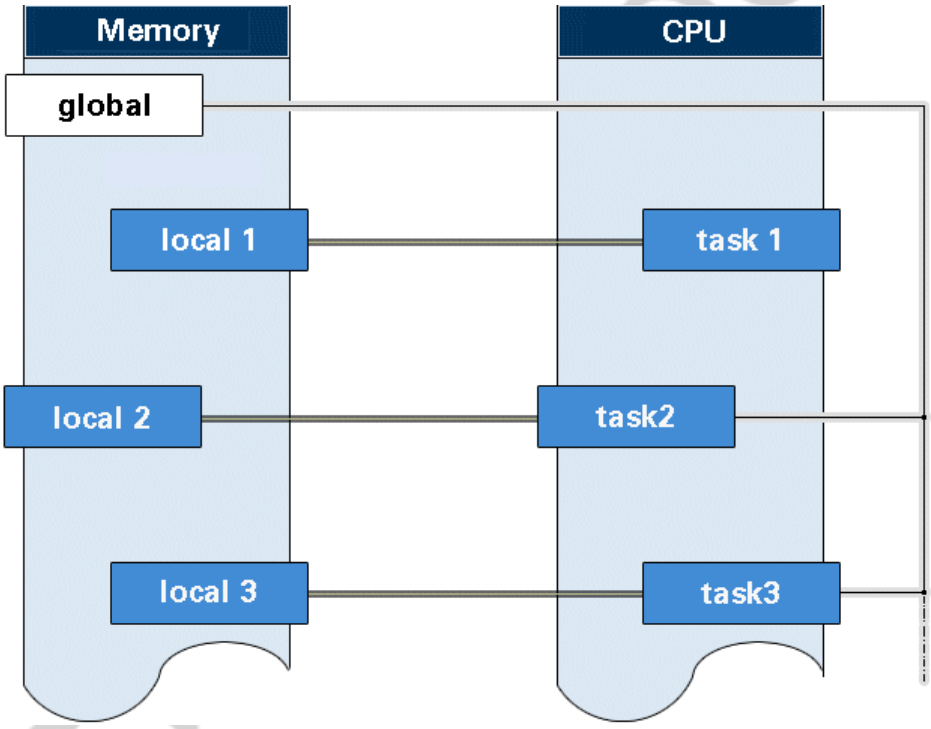


Fig. 6 Global / Local

3.3.2 Volatile, remanent / RETAIN and permanent variables

Volatile variables

Unless defined differently, variables are usually initialized during boot-up with the values configured when the variable is declared. In other words, they're volatile.

Remanent / RETAIN variables

It is often necessary for variable values to remain even after a system restart (warm restart). Variables that behave in this manner are called remanent variables.

If power is lost, a special logic operation copies remanent / RETAIN variables to a battery-backed memory area (SRAM) and back to their original location after a restart.

Permanent variables

Like remanent / RETAIN variables, permanent variables are also copied back after a system restart. Unlike remanent variables, however, they can also be restored after a cold restart.

Note:

Remanent / RETAIN or permanent variables should be used only when absolutely necessary. There are other ways to store data securely, e.g. data objects or files.

Task: Variables in Automation Studio 2.x

To learn more about working with local & global variables and constants, create an ST program that is divided into two tasks. Create the logical procedure for a press in the "press" task. When the "btnStart" button is pressed, the "doCylDwn" output should be controlled until the "diCylDwn" input is activated. This input should raise the cyclic counter "gCyclCnt". When the "btnStart" button is released, the "doCylUp" output should be controlled until the "diCylUp" input is activated. **Local** variables are used for the inputs and outputs. The "gCyclCnt" variable is **global**.

The code could look like this:

```
(* cyclic program *)
doCylDwn:= btnStart AND NOT(diCylDwn);

gCyclCnt:=gCyclCnt + EDGEPOS(diCylDwn);

doCylUp:= NOT (btnStart) AND NOT(diCylUp);
```

Fig. 8: Source code for the "press" task

The variable declaration should like this:

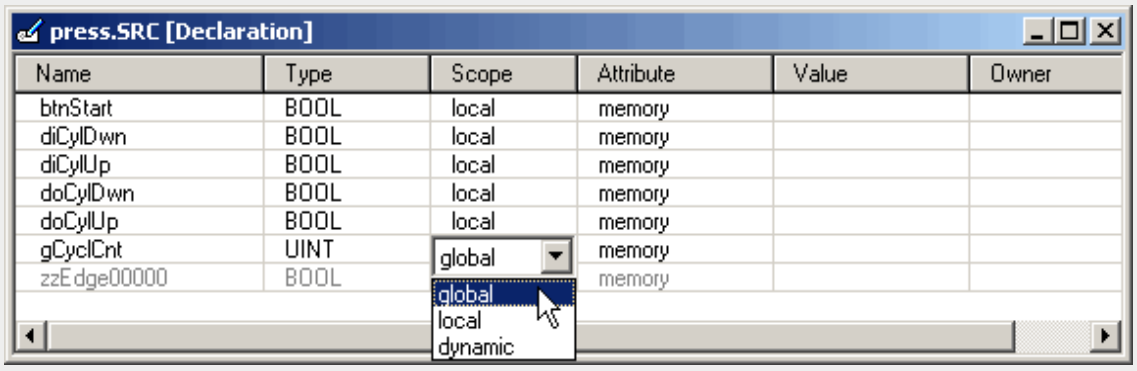


Fig. 8: Declaration for the "press" task

A **global** variable, "gLubricatet", is set in a second task, "mon", if the cyclic counter is larger than the constant "LUBR_CNT".

```
(* cyclic program *)
IF (gCyclCnt> LUBR_CNT) THEN
    gLubricate:= TRUE;
END_IF
```

Fig. 11: Source code for the "mon" task

mon.src [Declaration]

Name	Type	Scope	Attribute	Value	Owner	Remark
LUBR_CNT	UINT	global	constant	500		
gCyclCnt	UINT	global	memory			
gLubricate	BOOL	global	memory			

Fig. 11: Declaration for the "mon" task

When this program is transferred to the controller and started, the cyclic counter will be incremented with each cycle of the press. However, "gCyclCnt" is reset to zero each time the system is restarted or each time the task is downloaded because it is declared as a volatile variable. Modify the program so that this variable is maintained after a restart. To do this, you must enter "remanent" in the **Value** column of the declaration.

mon.src [Declaration]

Name	Type	Scope	Attribute	Value	Owner	Remark
LUBR_CNT	UINT	global	constant	500		
gCyclCnt	UINT	global	memory	remanent		
gLubricate	BOOL	global	memory			

Fig. 11: Remanent variable declaration for the "mon" task

The cyclic counter will now be modified so that it is maintained after a cold restart. To do this, it must be located in the permanent memory.

This can be done as follows:

Select the **Permanent** tab in the software tree and add the "gCyclCnt" variable via the shortcut menu **Variables:Insert Variables**.

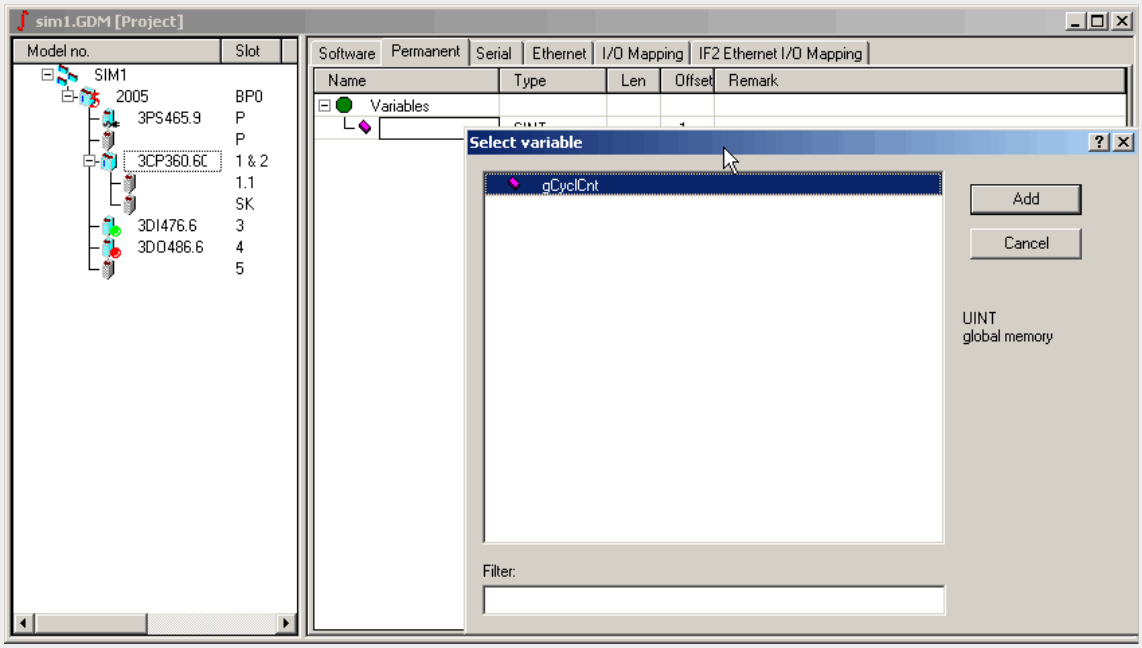


Fig. 12: Adding permanent variables

The offset for all configured permanent variables is calculated via the shortcut menu **Variables:Recalculate**.

Another permanent memory area must be configured before you can transfer the project.

To do this, open the shortcut menu **CPU:properties** from the **Software** tab in the software tree. Select the **Memory** tab.

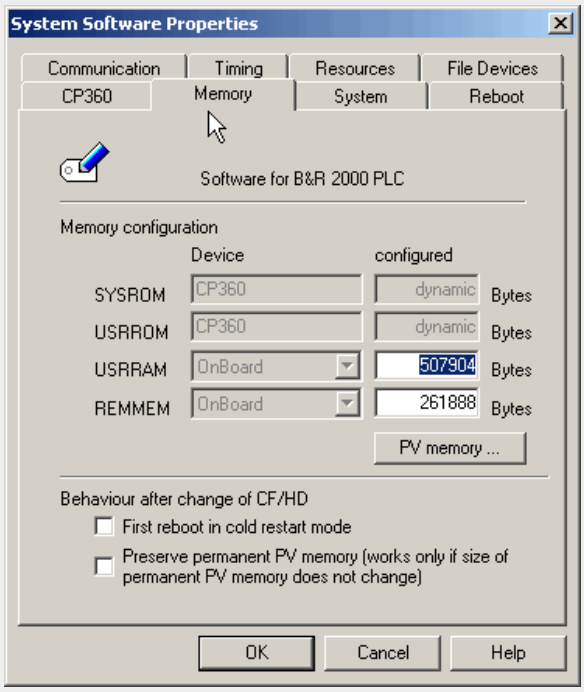


Fig. 14: Memory configuration

The following window is opened by clicking the button "PV Memory"

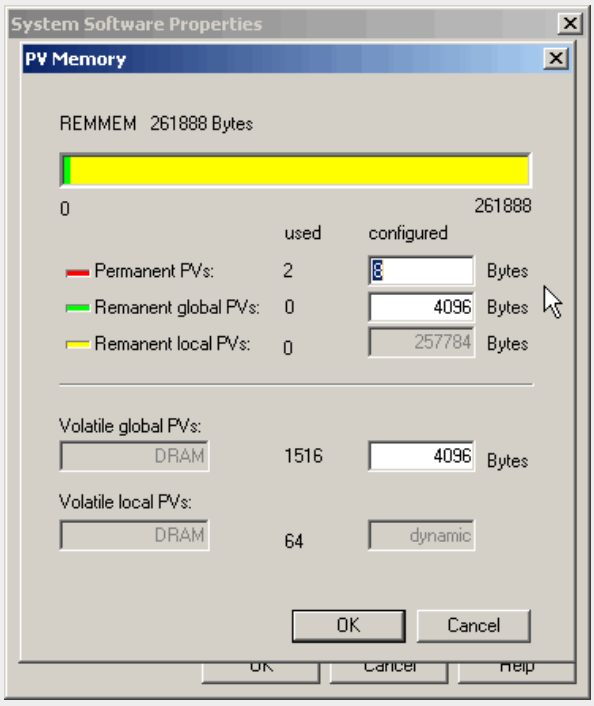


Fig. 14: PV memory configuration

Enter the number of required bytes for the permanent memory area. Click OK and retransfer the project.

Task: Variables in Automation Studio 3.x

To learn more about working with local & global variables and constants, create an ST program that is divided into two tasks. Create the logical procedure for a press in the "press" task. When the "btnStart" button is pressed, the "doCylDwn" output should be controlled until the "diCylDwn" input is activated. This input should raise the cyclic counter "gCyclCnt". When the "btnStart" button is released, the "doCylUp" output should be controlled until the "diCylUp" input is activated. **Local** variables are used for the inputs and outputs. The "gCyclCnt" variable is **global**.

The code could look like this:

```
(********************************************)
(* Implementation of program press          *)
(********************************************)

PROGRAM _CYCLIC

    doCylDwn := btnStart AND NOT (diCylDwn);

    gCyclCnt := gCyclCnt + EDGEPOS(diCylDwn);

    doCylUp := NOT(btnStart) AND NOT(diCylUp);

END_PROGRAM
```

Fig. 16: Source code for the "press" task

The variable declaration should like this:

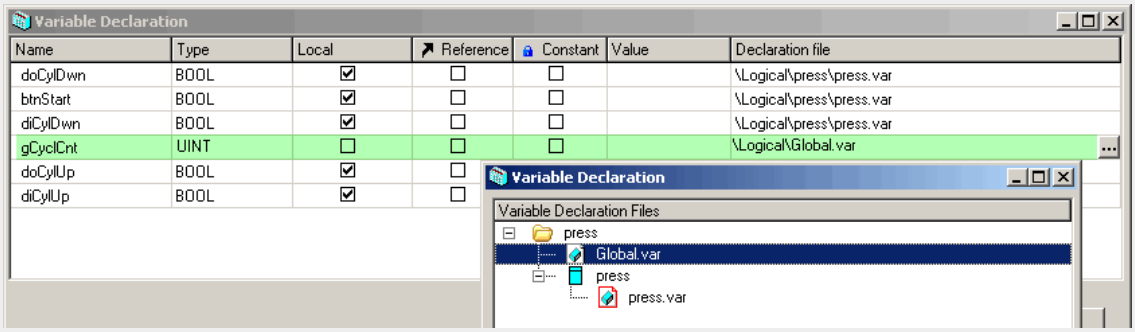


Fig. 16: Declaration for the "press" task

A **global** variable, "gLubricate", is set in a second task, "mon", if the cyclic counter is larger than the constant "LUBR_CNT".

```
(*********************************
* Implementation of program mon
*********************************)

PROGRAM _CYCLIC
    IF (gCyclCnt > LUBR_CNT) THEN
            gLubricate := TRUE;
    END_IF
END_PROGRAM
```

Fig. 19: Source code for the "mon" task

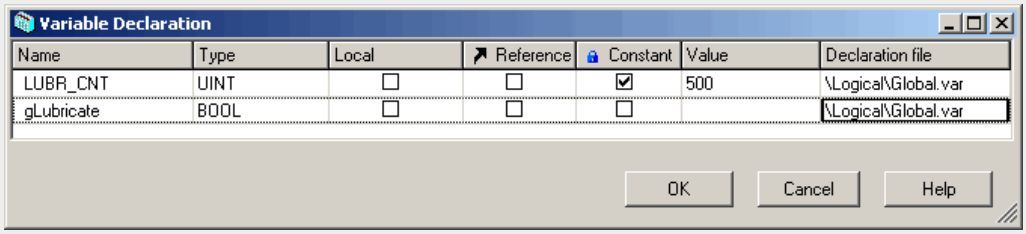


Fig. 19: Declaration for the "mon" task

When this program is transferred to the controller and started, the cyclic counter will be incremented with each cycle of the press. However, "gCyclCnt" is reset to zero each time the system is restarted or each time the task is downloaded because it is declared as a volatile variable. Modify the program so that this variable is maintained after a restart. To do this, you must enter "RETAIN" in the **Value** column of the declaration.

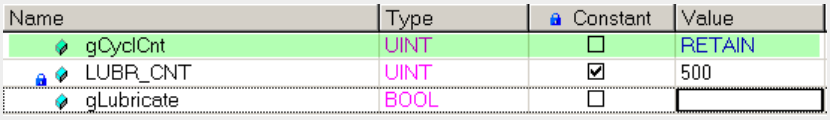


Fig. 19: RETAIN variable declaration

The cyclic counter will now be modified so that it is maintained after a cold restart. To do this, it must be located in the permanent memory.

This can be done as follows:

Select the shortcut menu of the cpu in the Physical View **shortcut menu:Open Permanent Variables** add the "gCyclCnt" variable via the shortcut menu **Variables:Append Variable**.

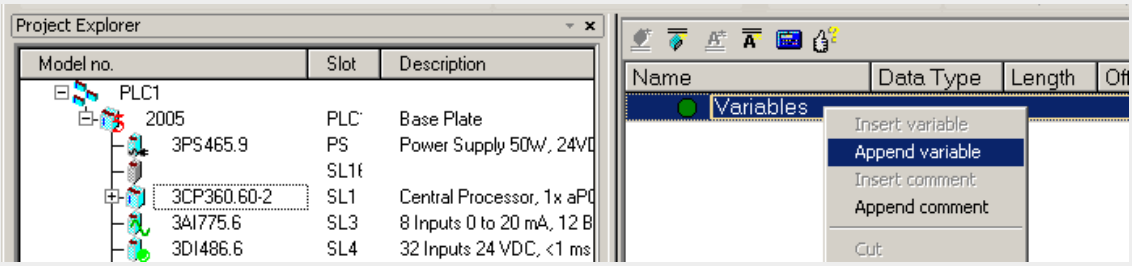


Fig. 20: Adding permanent variables

The offset for all configured permanent variables is calculated via the shortcut menu **Variables:Recalculate**.

Another permanent memory area must be configured before you can transfer the project.

To do this, open the shortcut menu **CPU:properties** from the Software Configuration tab in the software tree. Select the **Memory** tab.

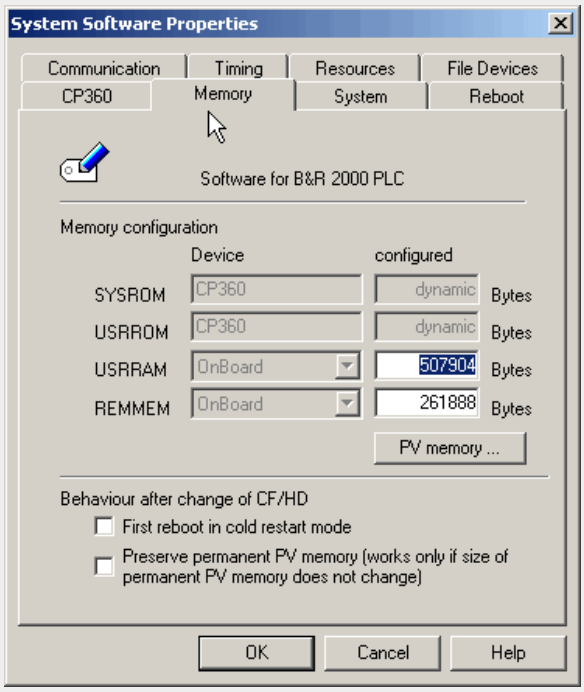


Fig. 22: Memory configuration

The following window is opened by clicking the button "PV Memory"

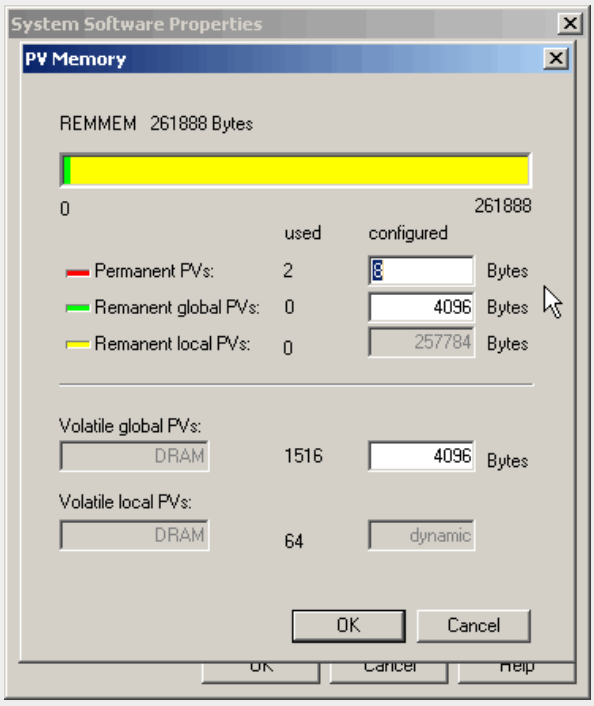


Fig. 22: PV memory configuration

Enter the number of required bytes for the permanent memory area. Click OK and retransfer the project.

3.3.3 Task download

When downloading a task, you can determine how the task variables should behave.

In most cases, the task is retransferred, variables are initialized with the value assigned when they are declared, the initialization subprogram (Init-Sp) is executed, and the cyclic part of the program is started. This is referred to as overload mode.

The following download modes are available:

- Overload:
  The value of the local variables is initialized. The Init subprogram is executed.

- Copy mode:
  The value of the local variables is maintained. No Init subprograms are executed. It could take several task cycles to copy the local variables.

- One cycle mode:
  Changes cannot be made to data types, number of variables, and existing variables; Init-Sp not executed. The task change is executed in one cycle.

Automation Studio 2.x

The download mode can be set by clicking on the **Advanced...** button under the **Transfer tab** after selecting the **Project: Settings...** entry in the main menu.

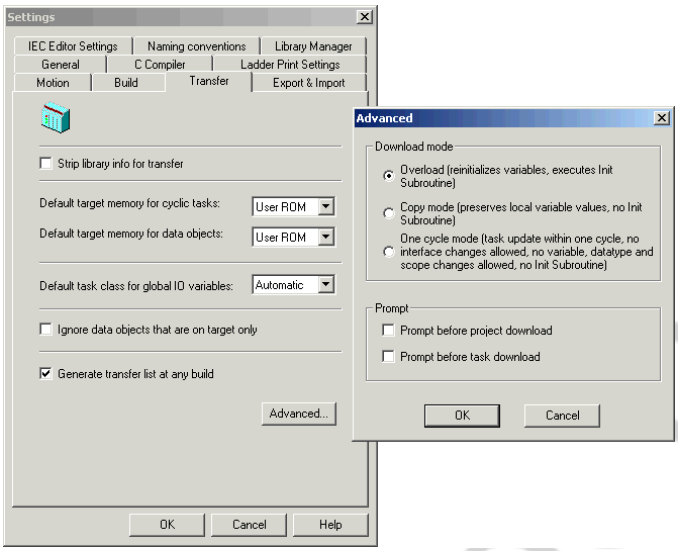


Fig. 23 Task download modes in AS 2.x

Automation Studio 3.x

The download mode can be set by clicking on the **Advanced...** button under the **Transfer tab** after selecting the shortcut menu of **PLC1:Properites** in the **Configuration View**.

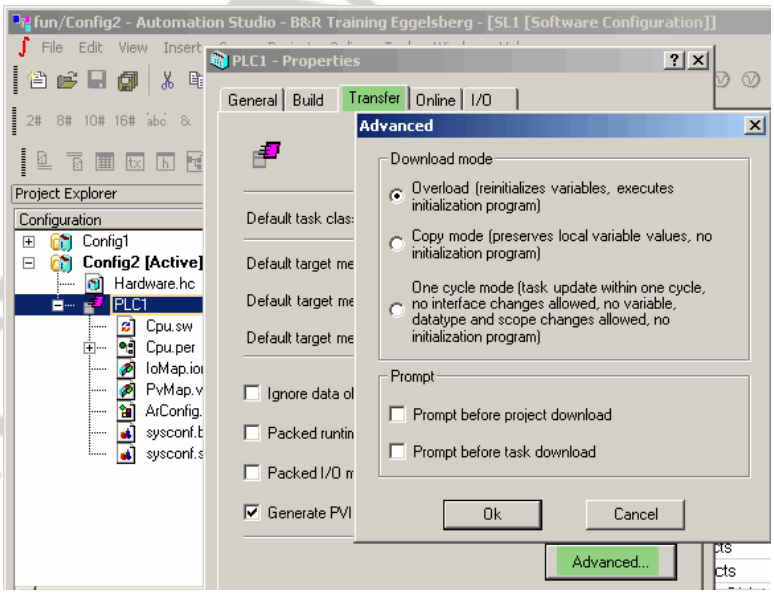


Fig. 24 Task download modes in AS 3.x

3.4 Buffer concept

Since a lot of data constantly changes, it is managed in a high-speed memory area. This data must be temporarily saved in the event of a power failure or warm restart. For this reason, there is buffering. Buffers can consist of rechargeable or normal batteries as well as gold foil capacitors. Most of the time they are located either on the module rack or directly built into the CPU. External buffering is also possible.

The status of battery buffering can be determined with the battery LED. In addition, the state of the battery can be checked from the controller program. It is often the case that the state is indicated on the visualization device. In any event, the backup battery should be replaced at certain maintenance intervals to guarantee maximum operational safety and data security.

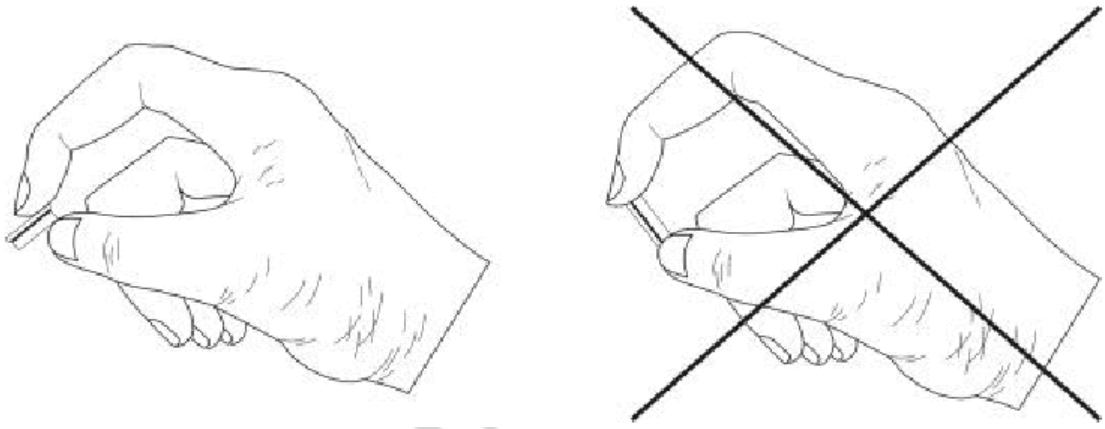


Fig. 25 Battery

The following areas / data are buffered:

- Real-time clock (RTC)
- SRAM (variable values)
- USER RAM (data objects)
- System RAM

Note:

Information about the lifespan of a battery can be found in the respective user's manual. How to change the battery is also described.

4. OPERATING STATES

The following operating states are possible after the controller is booted:

Mode	Description
RUN	After booting, all software modules to be executed are copied to DRAM and started. The controller runs, i.e. all programs are executed. Beforehand, the system was started with the configured boot cause (usually warm restart, see Boot Causes).
SERVICE	After booting, the task class system is not started. All activities (download, upload logger module, ...) can be carried out, as in RUN mode.
DIAGNOSTICS	This operating state only allows modules to be deleted from the target system, the logbook to be read, memory to be deleted, and a cold / warm restart to be carried out. Only modules from SystemROM are loaded. Memory can only be erased in diagnostics mode.
BOOT	Allows an operating system download to be executed. CompactFlash can be formatted and partitioned.

Note:

If you save the system configuration (Sysconf) to System ROM, then the connection settings for the interfaces are loaded in diagnostics mode as well.

4.1 Boot causes

The various boot causes can either be triggered **by the user**, **library functions**, or **by the operating system itself**.

The following table lists the different boot causes:

Mode	Description
Warm restart Loss of power	Buffering allows **remanent** data to be **kept**. A warm restart can be triggered by Automation Studio, PVI Transfer, pressing the reset button, or issuing a command in the controller application.
Cold restart	The complete **RAM** memory is **deleted** except for the permanent memory. The permanent variables are kept. A cold restart can be triggered by Automation Studio™, PVI Transfer, pressing the reset button, or issuing a command in the controller application.

4.1.1 RUN

This operating state is put into effect as follows:

- Cold or warm restart, loss of power
  The boot behavior when a power loss occurs can be set in the system configuration.
- Reset button on the CPU
- Library function

4.1.2 SERVICE

This operating state is put into effect as follows:

- Reset button on the CPU
- Stopping the target system with Automation Studio
- Error detected by the system at runtime
  e.g. cycle time violation, division by 0, etc.

Leaving service mode is done by performing a warm or cold restart.

4.1.3 DIAGNOSTICS

This operating state is put into effect as follows:

- Reset button on the CPU
- Mode selector switch
- Diagnostics mode with Automation Studio.
- Fatal system error

Leaving diagnostics mode is done by performing a warm or cold restart.

4.1.4 BOOT

This operating state is put into effect as follows:

- Requirement: CPU with default AR
- Mode selector switch
- No CompactFlash inserted
- No operating system present on CompactFlash

Leaving boot mode is done by performing a warm or cold restart (set mode selector switch correctly).

Note:

The boot behavior after a power failure or after pressing the reset button can be configured in the system configuration. Warm restart, cold restart, service and diagnostics are all possible settings.

4.2 Boot behavior

The following table shows the relationships between the boot causes and operating states:

	Boot causes		Operating states			
	Warm restart	Cold restart	Run	Service	Diagnostics	Boot
Deletes outputs	x	x	x	x	x	x
Deletes SRAM		x	(x)		(x)	(x)
Deletes PVs	x	x	x	x	x	x
Deletes remanent PVs		x	(x)		(x)	(x)
Recognizes HW system modules	x	x	x	x		
Sets PVs to Init value	x	x	x			
Starts task classes	x	x	x			
Cyclic processing	x	x	x			
Transfers application			x	x		
Starts system monitor			x	x	x	
Reads logbook			x	x	x	x
Performs warm restart			x	x	x	x
Performs cold restart			x	x	x	x
Clear memory...			x	x	x	
Transfers OS			x	x	x	x
Formats or partitions CompactFlash						x

(x) ... Depends on the configured boot cause

Boot causes are shown in blue. Yellow indicates the operating states, and the availability of service functions is shown in green.

5. MULTITASKING

Multitasking refers to software technology that makes it possible to both start several programs at the same time as well as for the computer to run certain program sequences in the background. The programs take turns using the CPU's processing power for this, but it goes so fast that the user gets the impression that everything is being done simultaneously.

The word itself is put together from "multi", a prefix meaning many, and "task", or something that is being carried out.

Automation Runtime is a deterministic real-time multitasking operating system. That means that al tasks are executed in a fixed, predictable time frame.

5.1 Task

A task refers to something that needs doing. In this case, its contents represent an independent controller task. A complete application is usually made up of several tasks that work together.

5.2 Task properties

A task has several properties that are determined by the system and can be set by the user.

A task has the following properties:

- Description
- Date / time
- Version number
- Target memory
- Compiler options (back-end options)
- Priority (task class)
- Checksum (monitored by the operating system)

5.3 Task classes

Not all tasks have to run within the same timeframe. For many controller tasks, it's extremely important that they run very fast and very often (e.g. loop controller). For others, it's OK if they aren't processed as often (e.g. output for visualization).

The task classes enable a task to optimally meet its requirements. A task class comprises of tasks with the same cycle time. The task class is used to define the priority, cycle time and tolerance time.

The number of task classes and their default settings is different depending on the target (processor capacity). The default settings for a CP360 are listed in the following table.

Task classes and default values:

Task class	Cycle time [ms]	Tolerance [ms]
Cyclic #1	10	10
Cyclic #2	20	20
Cyclic #3	50	50
Cyclic #4	100	100
Cyclic #5	200	100
Cyclic #6	500	100
Cyclic #7	1000	100
Cyclic #8	10	3000

The following image shows the correlation between executing one or more tasks within a task class and available resources.

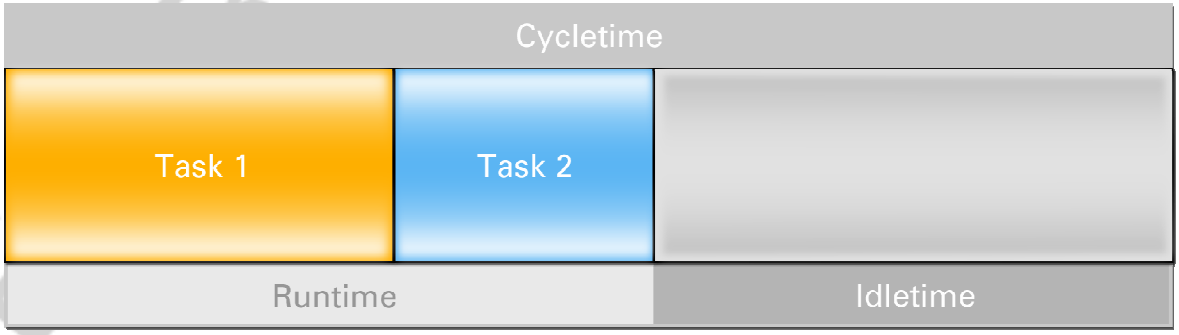


Fig. 26 Runtime, idle time, cycle time

Each task class has a predefined **cycle time**. All of the tasks in a task class must be executed within this amount of time. The time it takes for a task to be executed is called its **runtime**. The sum of all task runtimes must be less than the cycle time of the task class. Any leftover time is called **idle time**.

Task: Cycle time

Create a task in which the status of an output changes with each cycle.

The code could look like this:

```
(* cyclic program *)
doOut1:= NOT doOut1;
```

Fig. 28: Code

Name	Type
doOUT1	BOOL

Fig. 28: Declaration

Run the task once in the 20ms, the 100ms and a 700ms task class.

The task class can be set to a task in the shortcut menu **properties**.

5.3.1 Tolerance

The tolerance is reached when the sum of all task runtimes in a task class exceeds the cycle time. A task class's **tolerance** is primarily used to intercept sporadic cycle time violations. However, this has the effect of pushing the start of the new task class back, which is generally not desirable since it creates timing problems in the application.

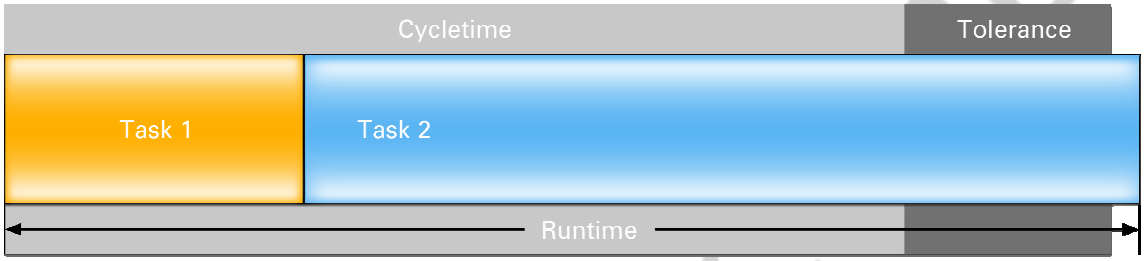


Fig. 29 Exceeding cycle time + tolerance = cycle time violation

A cycle time violation is when the cycle time of the task class and the tolerance is exceeded. This is a very dangerous state and causes the system to restart in service mode.

In all cases, you must check as to whether the task runtimes are permitted for the respective task class in order to guarantee that the application can be executed reliably.

Task: Cycle time violation

Create a task with a loop whose final value is a variable. Increase the final value until a cycle time violation occurs. Take a look at the error message in the Logger. Try this in different task classes.

The code could look like this:

```
(* cyclic program *)
FOR i:=0 TO endValue DO
    value:= value + 1;
END_FOR
```

Fig. 31: Source code

Name	Type
value	UDINT
i	UDINT
endValue	UDINT

Fig. 31: Declaration

5.4 Scheduling

Several tasks in a project are executed more or less at the same time. Coordinating the execution of these tasks is handled by what is called the I/O scheduler.

The I/O scheduler

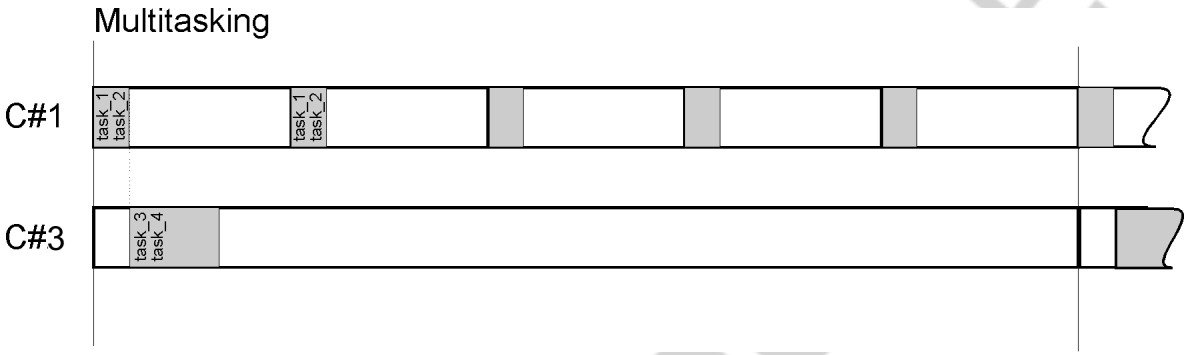


Fig. 32 I/O scheduler

The I/O scheduler activates the respective tasks when a task class is started and provides them with the I/O image.

At the beginning of each cycle, the following checks are made:

- Whether the tasks have finished within the configured time
- Whether the new input image was received completely
- Whether the output image has been written

5.4.1 Idle time

The following graphic compares task class Cyclic #1 (10ms) and Cyclic #4 (100ms). A task with 9 ms runtime is used once in Cyclic #1 and once in Cyclic #4.

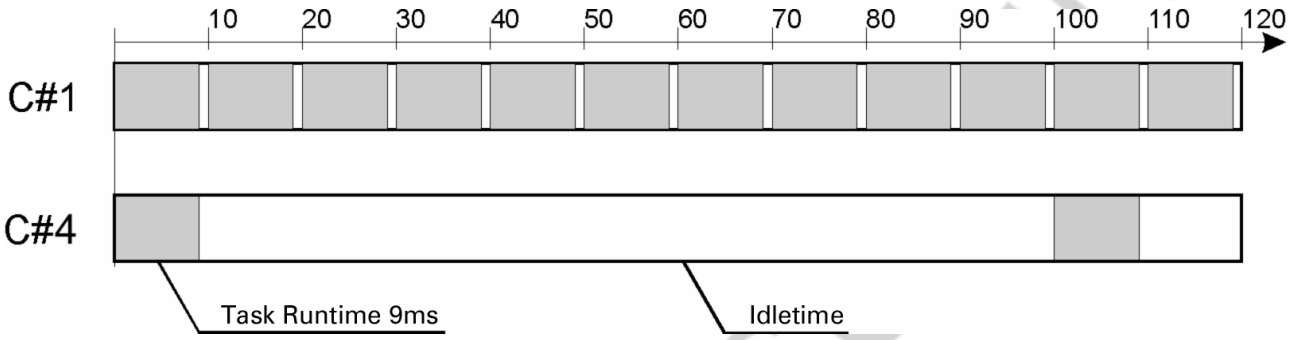


Fig. 33 Task runtimes

The following points can be seen in the diagram above:

Task class	Processing time in 100 msec	Idle time per process [ms]	System utilization in %
Cyclic #1	10	1	90
Cyclic #4	1	91	9

This graphic clearly shows the influence that the task class selection can have on the system load. The distribution of the individual tasks into different task classes, which sufficiently meet the application's requirements, is an important topic.

The more idle time that remains on a system, the more time resources are available for expansions.

The following tasks are taken care of by the operating system in the idle time:

- Online interface communications
- Visualization operation and animation
- Checksum monitoring for all modules on the target system
- File access

5.4.2 Idle time configuration

This can be defined in the system configuration so that there is enough idle time for communication when the CPU is under a high load. The idle time is assigned to a task class, where it is placed at the end.
At the beginning of the idle time, the priority of the task class where the idle time has been configured and all underlying task classes is reduced (e.g. if the idle time is specified in Cyclic #2, then Cyclics #3, 4, etc. are affected).
All idle time services (e.g. communication, etc.) have a higher priority and are processed by the system. Once the idle time expires, the original priorities of the task classes are restored.

The idle time configuration can be opened in the software tree / software configuration by selecting the **Timing tab** from the **Properties... shortcut menu entry**.

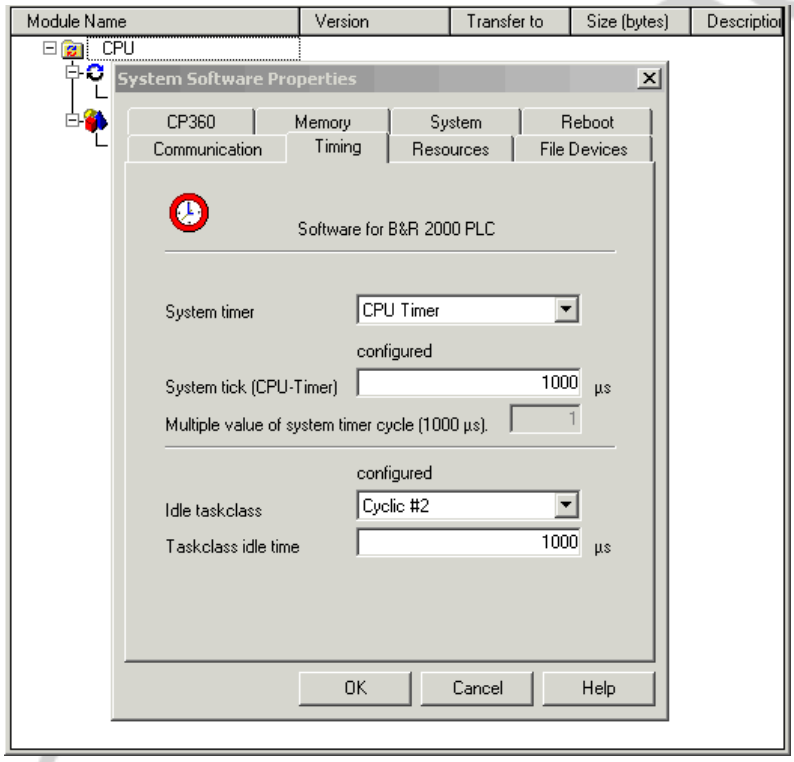


Fig. 34 Idle time configuration

Note:

Higher priority task classes are not affected by the temporary change of priority at the beginning of the idle time (e.g. idle time in Cyclic #2 does not affect Cyclic #1).

5.4.3 Multitasking mode

We will use an example to demonstrate how multitasking takes place on the controller.

Tasks that have varying runtimes are placed in several different task classes; tasks that have a longer runtime are put into task classes with a longer cycle time.

Runtime [ms]	Resource	Task class cycle time [ms]	Tolerance [ms]
0.8	Cyclic #1 (C#1)	10 ms	10 ms
1.6	Cyclic #2 (C#2)	50 ms	50 ms
20.0	Cyclic #3 (C#3)	100 ms	100 ms
2.2	Cyclic #4 (C#4)	10 ms (as fast as possible when the system has free time)	1000 ms

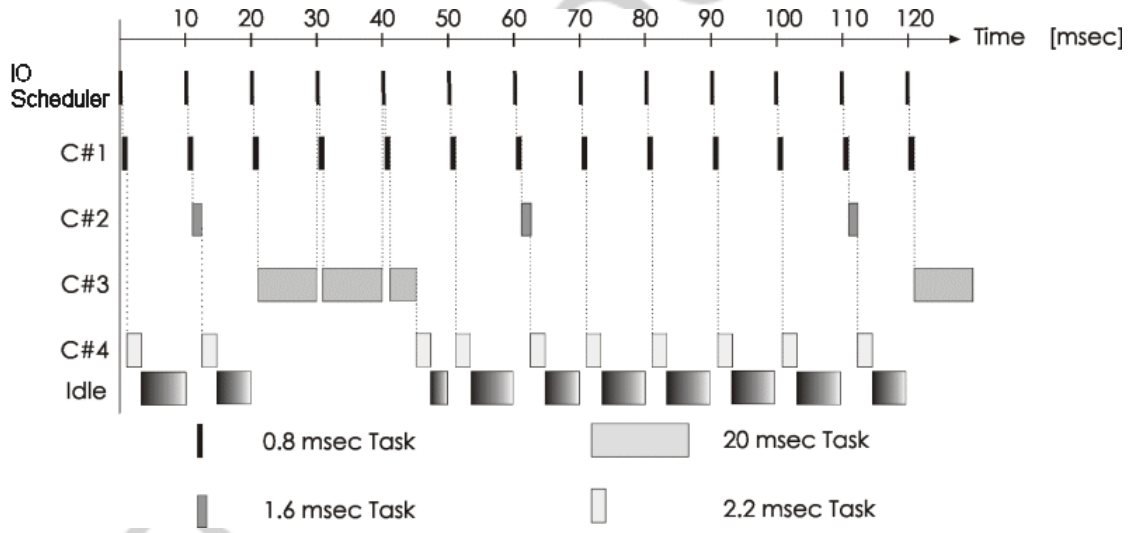


Fig. 35 Multitasking

The following behavior is depicted in the image:

Task classes have different **priorities**. These priorities go down with the number of the task class.

The **Cyclic #1** task class has the **highest** priority. After that follows Cyclic #2, etc. The Cyclic #4 task class is executed as often as possible, but only if there is time still available.

When the system is booted, the task classes are started beginning with those that have the highest priority. After Cyclic #1 starts, there is already available time, which allows Cyclic #4 to start.

Task classes with lower priority are interrupted by task classes with higher priority.

After Cyclic #2 starts, there is once again time remaining for Cyclic #4. Then Cyclic #3 starts, is **interrupted** by Cyclic #1, and then **continues** for as long as it takes until the task class is finished. In the meantime, there is no available time left for Cyclic #4.
Now all task classes have been started, and multitasking mode is in effect.

The nearly simultaneous execution of tasks in a multitasking operating system that we talked about earlier is now a little bit clearer.

More or less at the same time means that the different priorities may lead to some interruptions.
This makes it possible for each task class to be processed in its own time slot even if task classes with lower cycle times are executed more often.

5.4.4 Real-time

Real-time refers to the ability of an operating system to provide services within a predictably limited response timeframe.

When transplanted to a multitasking system, this means that the I/O scheduler is always executed at an exact moment. The result is that task classes execution is started at a predictable moment, hence "deterministic".

Automation Runtime is a deterministic real-time multitasking operating system.

Several tasks can be executed more or less at an exactly predictable point in time.

5.5 Exceptions

Exceptions are fatal errors that occur during runtime and cannot be corrected by the operating system alone.

By default, the system carries out an emergency stop when an exception occurs, i.e. the controller boots in service mode.

In contrast to other errors, exceptions give the user the possibility not only to detect that an error or exception occurred, but also the opportunity to react to it. For this purpose, the operating system provides an **exception handler** which can handle the most common exceptions according to the application.

The exception task class has the highest priority in the system. When an exception occurs, the accompanying exception task interrupts all tasks.

The following errors can be detected by an exception task:

- Division by zero
- Cycle time violation

Note:

Although the errors listed above can be intercepted by exceptions, it's much more important to make sure that these errors never occur when the application is being created.

5.5.1 Inserting an exception task

An exception task must be enabled in the system configuration before it can be added to the project.

In the software tree / software configuration, click on the **Properties...** menu item from the shortcut menu, switch to the **Resources tab**, and enable the **"Enable exception task class"** option.

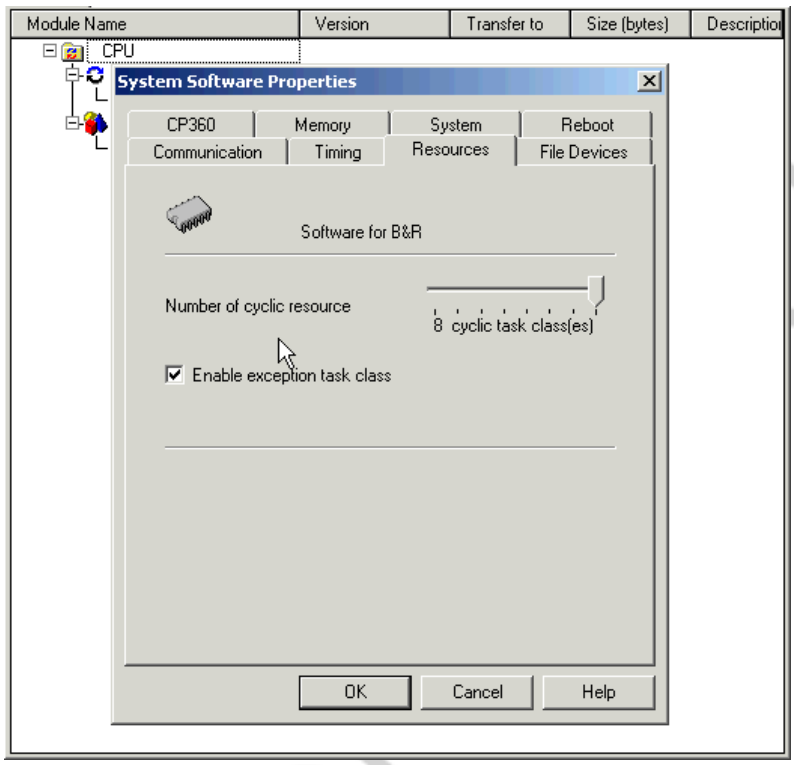


Fig. 36 Enabling the exception resource

An additional resource then is available in the software tree / software configuration when a task is inserted. To determine what the exception in the respective task corresponds to, select the **Properties...** entry in the **shortcut menu**.

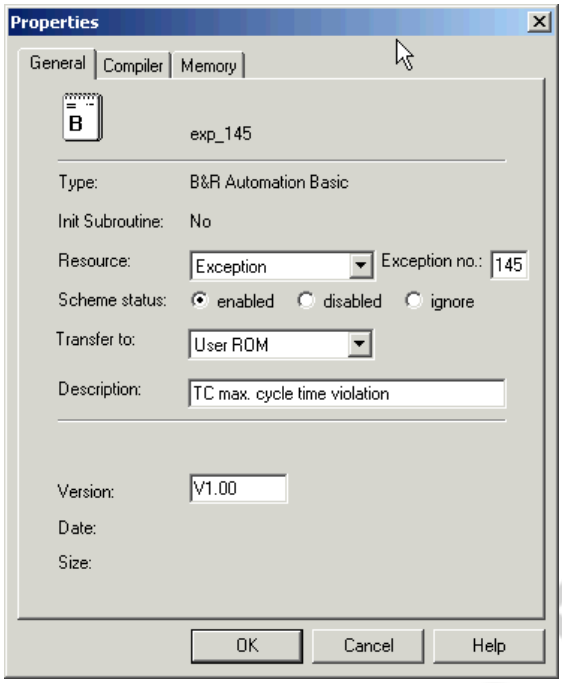


Fig. 37 Settings for an exception task

The **Exception no.** specifies the type of exception; information about it can be read in the Automation Studio™ online help documentation.

5.6 Task initialization

Each task can have its own initialization sub-program (Init-Sp). This sub-program allows task-specific initialization operations to be carried out.
The initialization subroutines of all tasks are executed one after the other when the controller is booted.
The controller only enters multitasking mode when the last Init-Sp is finished executing.
Since Init subprograms are **not time-monitored**, initialization phases that take a long time do not result in cycle time violations.

6. AUTOMATION RUNTIME I/O MANAGEMENT

One central function of a PLC system is being able to transport I/O states to or from an I/O terminal as fast as possible, both inside and outside of the application program.

B&R Automation Runtime I/O management makes it possible for I/O transport to meet the highest demands on response time, minimized jitter, and configurability.

6.1 General

Transferring data from a data point to the I/O terminal is the task of the I/O manager. The **I/O manager** is controlled by the **I/O configuration** and the **I/O mapping**. The application software is processed in **the task class system**. The I/O manager carries out its task both before and after a task class.

6.2 Basic functionalities

When the controller is booted, the active I/O configuration is transferred to the I/O modules. This guarantees that valid I/O data is already provided in the Init-Sp.

6.2.1 Principle

The I/O configuration makes sure that the interfaces and the fieldbus devices are initialized before the Init subroutine begins.

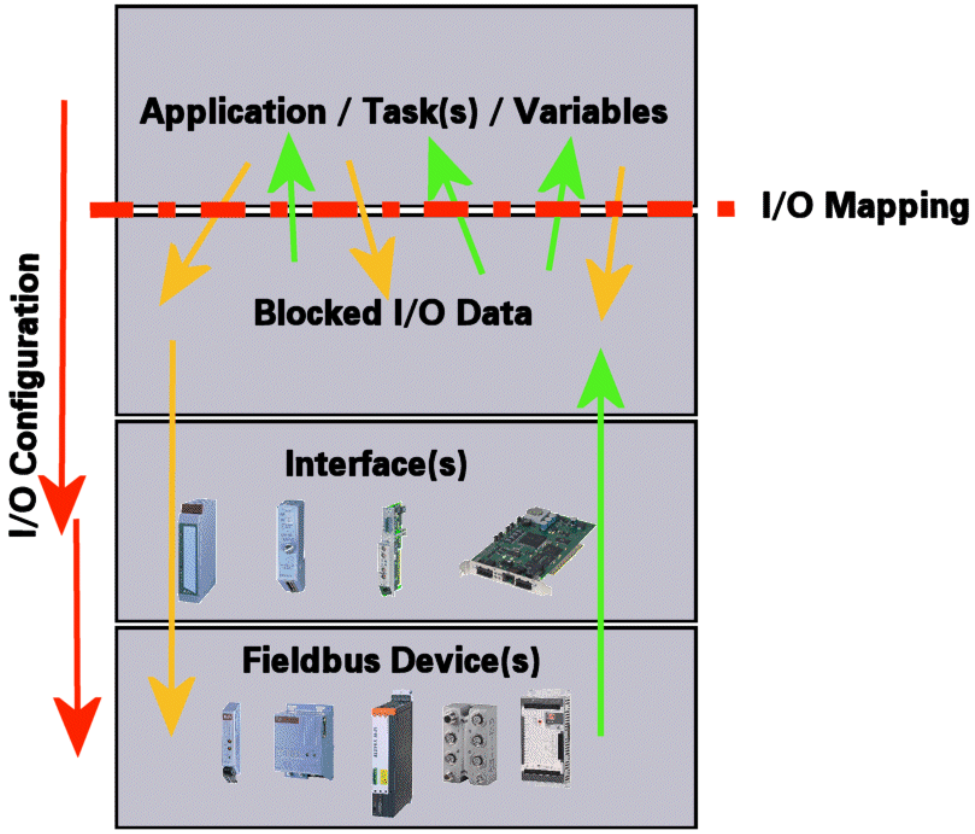


Fig. 38 Basic function of I/O handling I

Blocked input data from the fieldbus devices is stored in memory. The data is sorted into control variables using I/O mapping.
The output data is written in reverse order via the blocked output data.

The following graphic shows the described function in more detail.

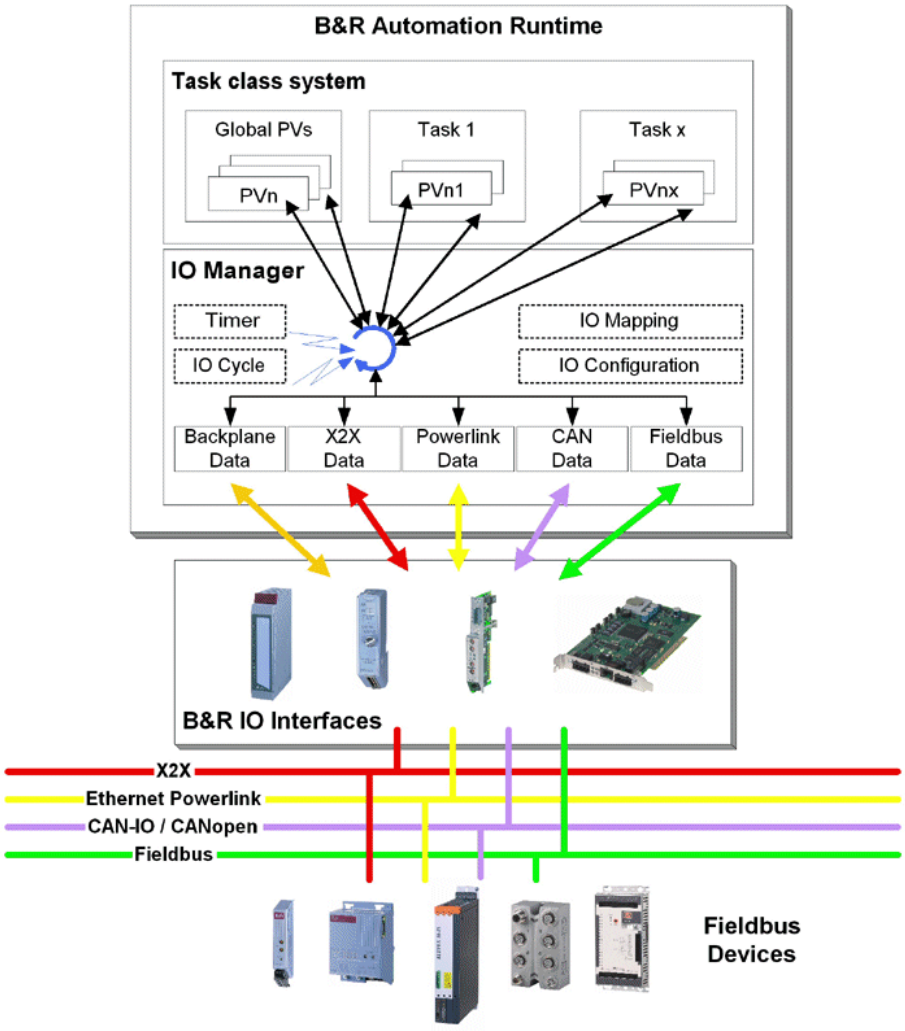


Fig. 39 Basic function of I/O handling II

6.2.2 I/O Mapping

I/O mapping determines which I/O data is transferred to or from which process variable. I/O mapping can be generated externally as well as during runtime.

Each variable that exists in an application (no C-internal variables) can be assigned to an I/O module channel regardless of the scope.

The desired I/O module must be selected to assign variables to the hardware. The right-hand side of the screen now shows the view for the selected I/O modules.

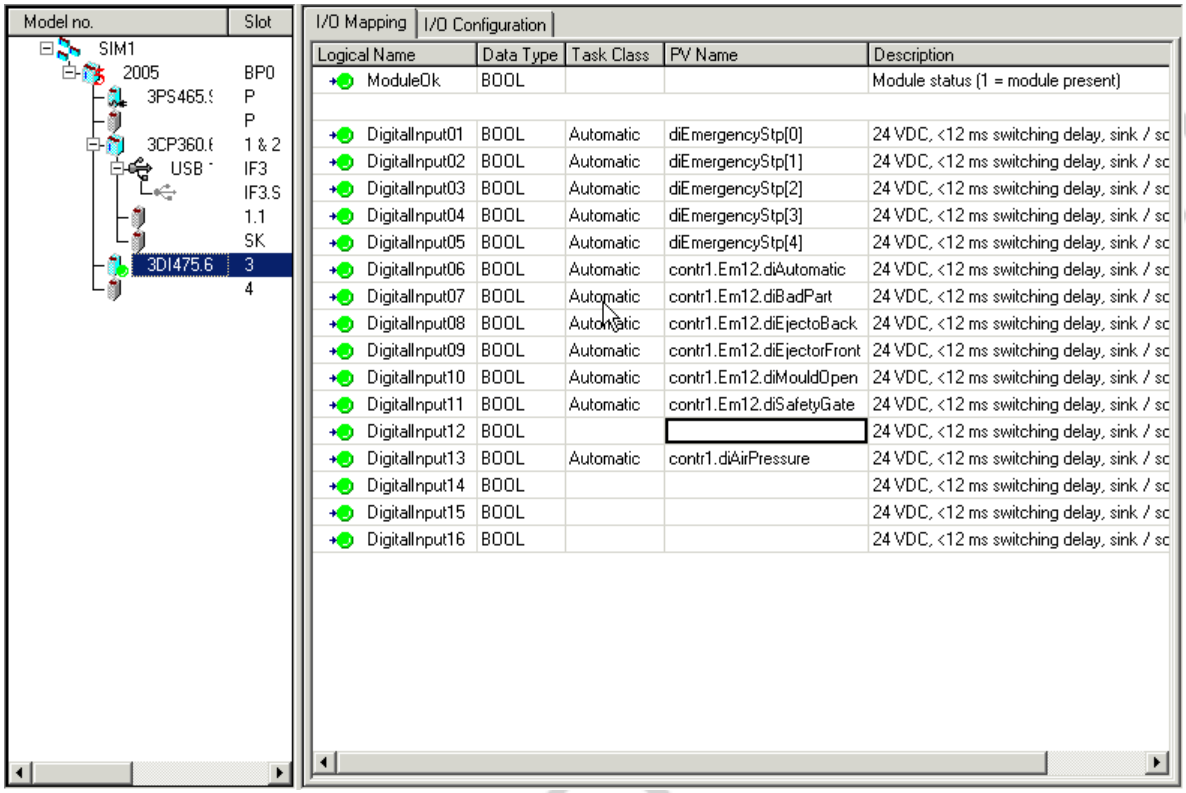


Fig. 40 Editing the I/O mapping

In addition to the **Logical name** and the **Data type** for the variables that can be connected, the **Task class** and **PV name** columns are also listed here.

For **global variables**, the respective **task class** in which the data transfer to or from the I/O image takes place must be configured in the **Task class** column. Selecting the task class as **Automatic** means that Automation Studio automatically determines the fastest task class where the variable is being used when the project is built.
Local variables are always updated in the task class of their task. Specifying an **automatic** task class is unnecessary for local variables and ignored by the compiler.

Note:

Specifying a cyclic task class classifies a variable as a global variable, i.e. the system assumes that the variable listed in the "PV name" column is a global variable. If the declaration for this variable is missing, then the compiler returns an error.

6.2.3 I/O configuration

Module-specific properties can be configured on the **I/O Configuration**. The I/O configuration can be generated externally as well as during runtime.

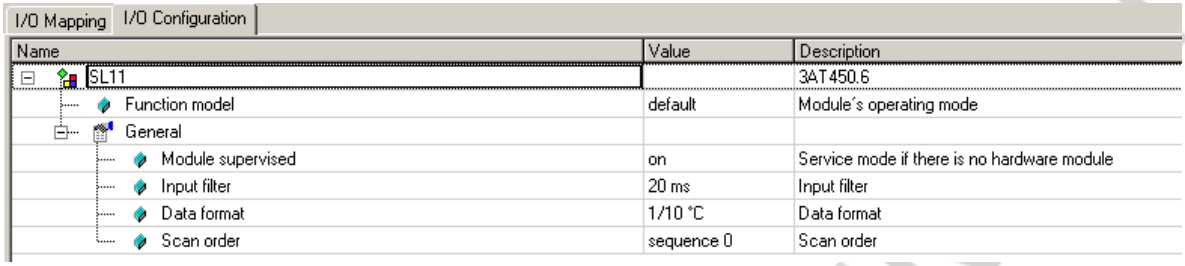


Fig. 41 Editing the I/O configuration

6.3 I/O handling in the task class system

The I/O manager handles or passes on the blocked I/O data of the I/O interfaces and forwards it to the process variables used in the tasks according to the I/O mapping. This process is also called singling out. The task class where the process variables are being used is taken into consideration.

Since a lower-priority class can be **interrupted** by a higher-priority class, it's necessary that a separate I/O image be provided each time a **task class** is started.

Inputs are transferred at the beginning of the task class; outputs are transferred at the end. Because of this, the input states remains the same during the task's runtime and the output state are only output at the end of the task class. The task class is always provided with only one **complete image**. That means that a check takes place at any time to see if an image arrives completely. If this is not the case, then the last image recognized as valid is used.

The I/O scheduler controls when task class data is prepared and when the task classes are started.

Note:

When creating the I/O mapping, inputs can be used more than once. An additional mapping can be added by opening up the shortcut menu for the respective channel.

Properties can be configured for task class #1 using the shortcut menu in the software tree, to determine whether the outputs should be written after the last task in a task class (fast-response writing) or at the end of the task class cycle (jitter-free writing).

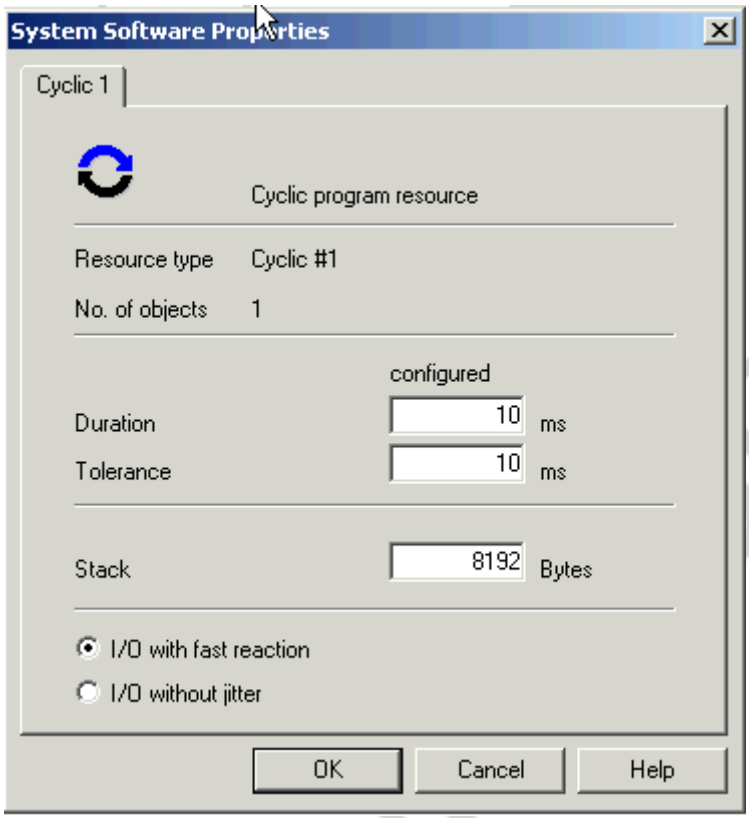


Fig. 42: Setting for fast-response or jitter-free writing of outputs

6.3.1 Fast-response writing of outputs

The outputs are written immediately after the last task in the respective task class when **I/O with fast reaction** is selected.

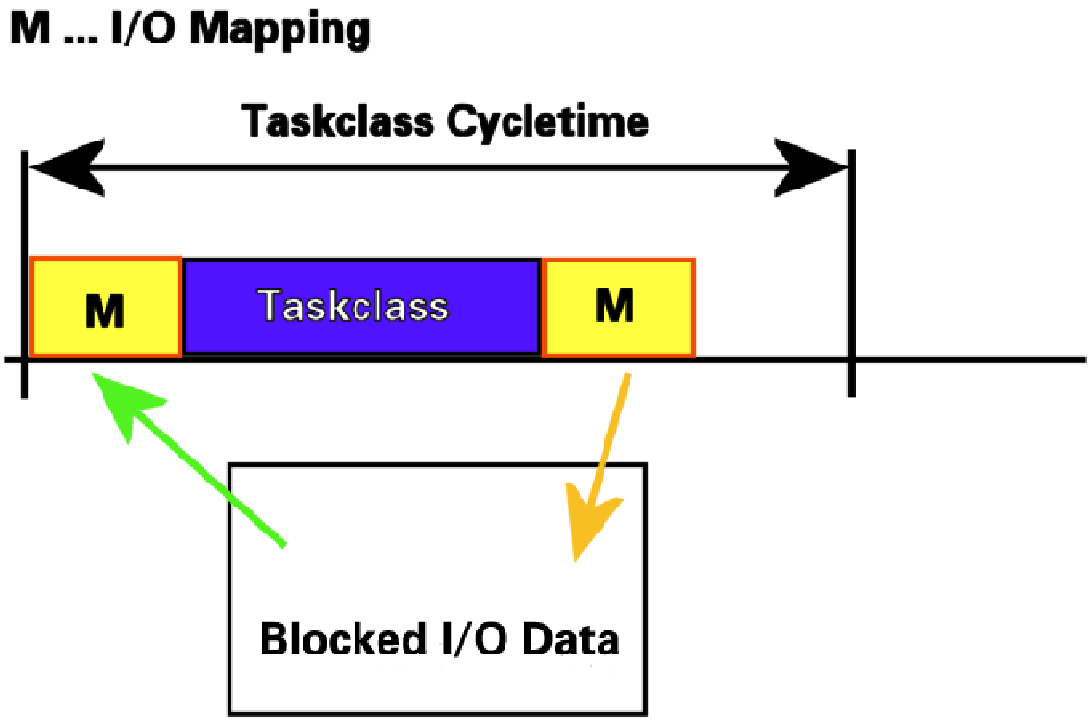


Fig. 43 Fast-response writing of output data

6.3.2 Jitter-free writing of outputs

Unlike the fast-response writing of outputs, the **I/O without jitter** option means that outputs are written at the end of the task class. In this way, the outputs are always written at the same point in time even when the individual tasks in the task class have different runtimes.

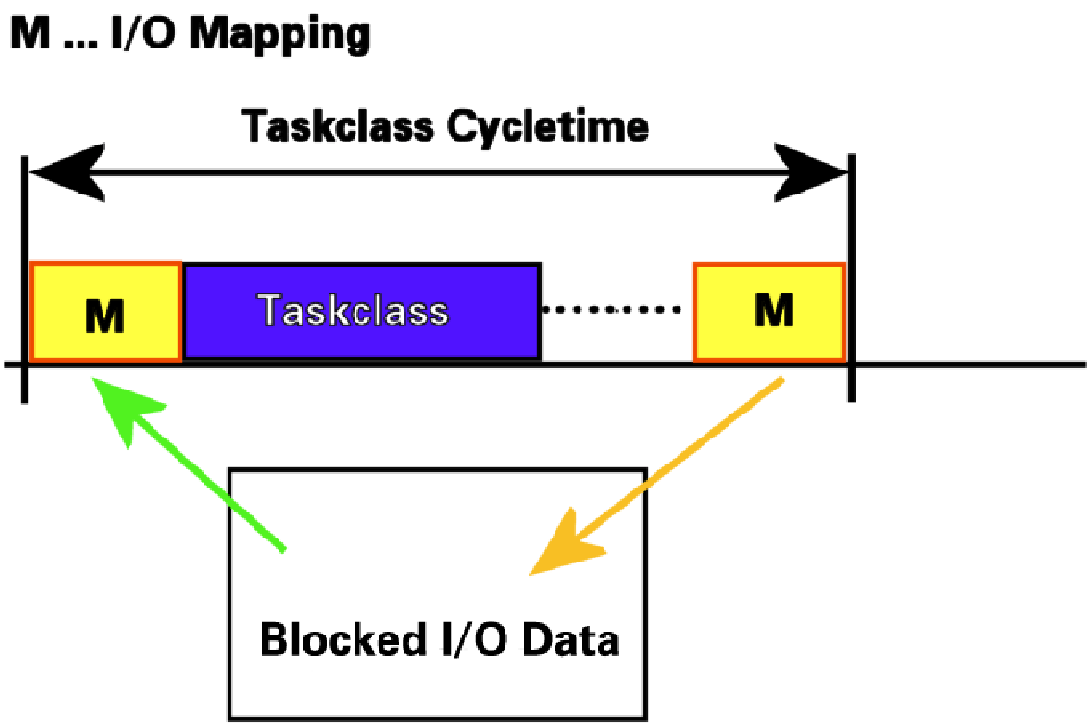


Fig. 44 Jitter-reduced writing of output data

6.4 External configuration

The B&R system provides the opportunity of creating the I/O configuration and I/O mapping without needing the Automation Studio user interface. This is called creating an external configuration.

In practice, configuration data is programmed using script languages or XML translations from product or machine configuration data stored in a company database.

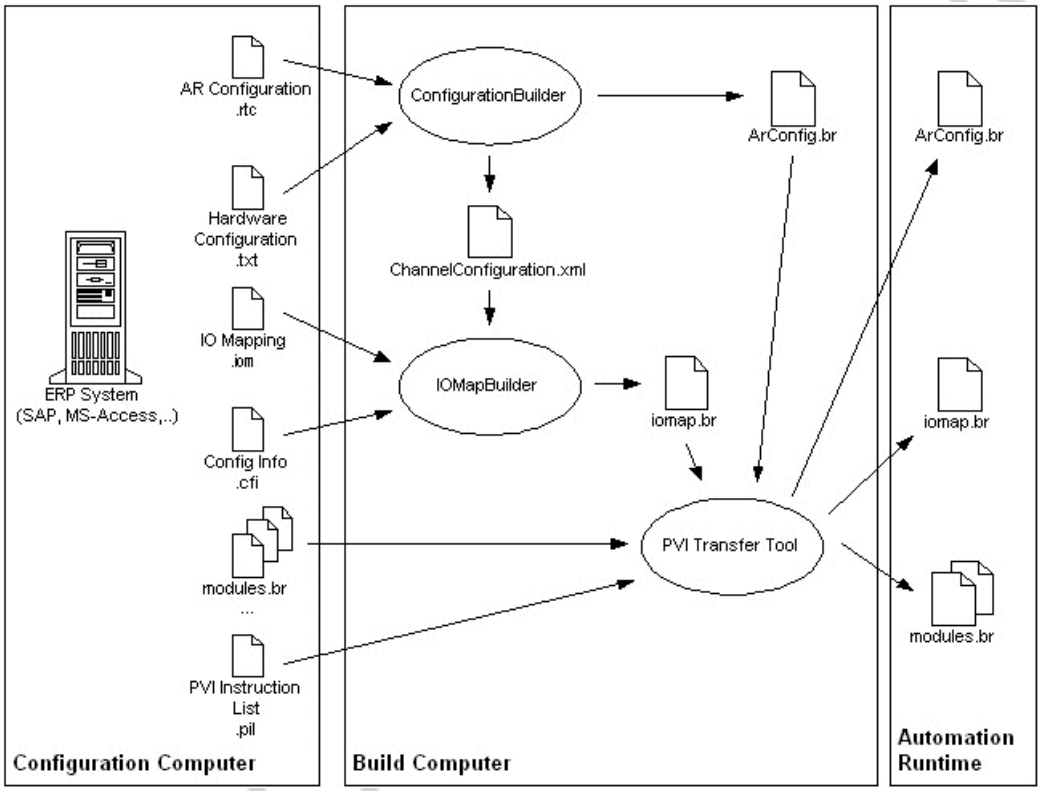


Fig. 45 External configuration

Once this source data has been created, the available compiler program can be used to generate the binary configuration files (.br modules), which can then be transferred to the respective controller using the PVI Transfer tool.

The **BR.AS.ConfigurationBuilder.exe** and **BR.AS.IOMapBuilder.exe** compilers are available to create I/O mapping and the I/O configuration without having to use the AS user interface.

Note:

More information about this topic can be found in the Automation Studio online help.

6.5 Configuration at runtime

I/O operation is generally based on the I/O configuration (arconfig.br) and the I/O mapping (iomap.br). Corresponding libraries (AsIOMMan) make it possible to manage different I/O configurations and I/O mappings as data objects on the target and enable them whenever needed (converting a data object to an I/O configuration module or an I/O mapping module).

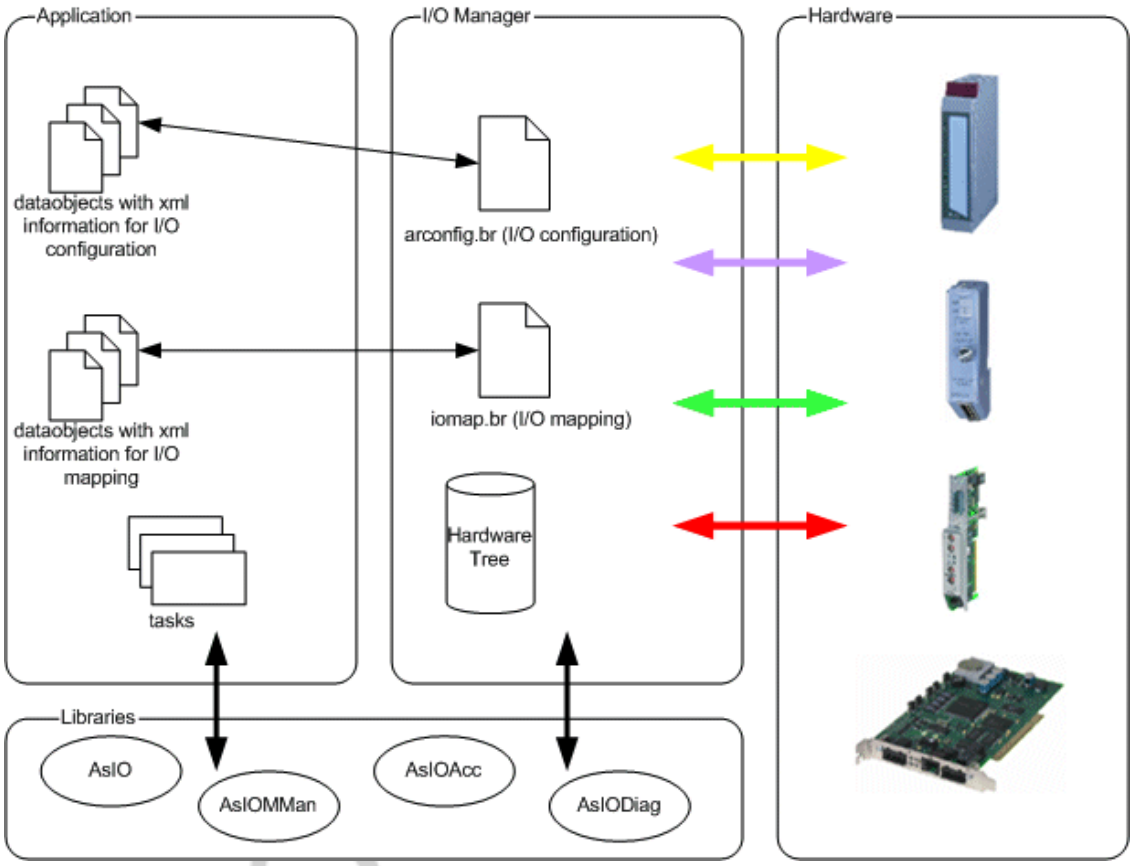


Fig. 46 Configuration at runtime

The configuration at runtime is thus put together with the following parts:

- Managing data objects with I/O configuration data
- Managing data objects with I/O mapping data
- Editing data objects using the DataObj library
- Activating or deactivating individual modules (converting a data object to an I/O configuration module or I/O mapping module).

Other libraries (e.g. AsIO, AsIOAcc, etc.) can be used to handle appropriate interactions between the application and the I/O manager, beginning with reading the hardware tree up to reading or setting individual I/O data points.

7. INSTALLATION AND UPDATES

There are various operations system versions available. The reason for this is that there are constantly new developments being made for this system. The most current operating system version is located on the Automation Studio CD. However, all older operating system versions can be installed with Automation Studio as well. This makes it possible to only have to update the project if new operating system functions are needed.

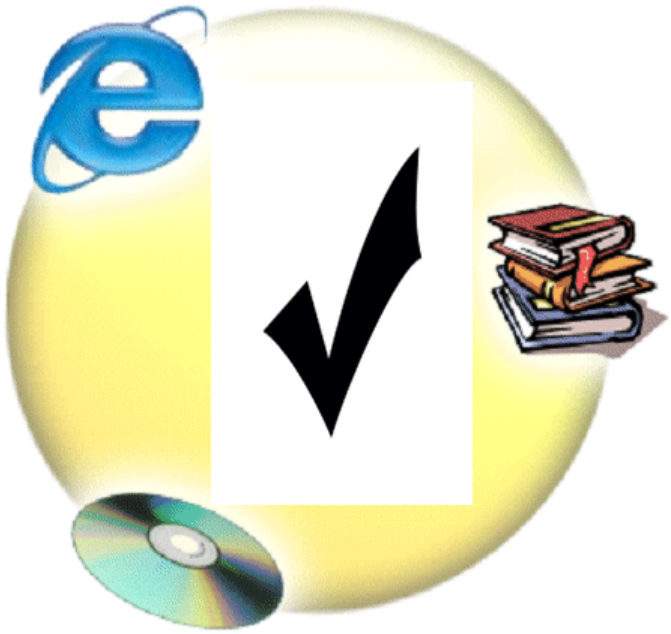


Fig. 47 Information

7.1 Upgrades

A few operating system versions are included on the Automation Studio CD. Newer operating systems, which are released at regular intervals, can be obtained from the download area of www.br-automation.com and then added to the existing Automation Studio installation.

There is also the possibility to upgrade hardware, operating system and parts of the Motion Components and Visual Components by clicking on **Tools**:**Upgrades**. The new components will be downloaded and installed from the homepage.

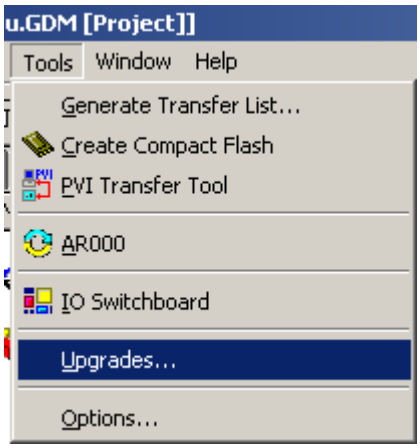


Fig. 48: Updating hard- and software

Differences between the individual operating systems can be found in the revision history in the download area.

The CPU is always delivered with a default operating system. From the time he gets it, the user can establish a serial online connection to carry out an operating system download. The default operating system is simply a minimal version of Automation Runtime.

The operating system and application are stored on CompactFlash if the system is using this for memory. The default operating system is stored in the CPU's internal memory.

There are essentially three ways to carry out an operating system download.

- Download or update in a running state, i.e. **RUN**, SERVICE, or DIAGNOSTICS.
- Update in **BOOT mode**. To switch to BOOT mode, you have to change the hardware selection switch from RUN to BOOT. More information about the individual CPUs can be found in their manuals. Only the default operating system is loaded in BOOT mode.
- The Compact Flash is created on the PC via the PVI Transfer Tool.

The operating system can be downloaded over a **serial interface** or by putting it on a CompactFlash card (for target systems that use **CompactFlash** as application memory).

Note:

All memory is erased when an operating system is updated. For this reason, all important data should be backed up beforehand.

7.2 Changing the operating system version

After you have installed the operation system on your PC as active Automation Studio version, it can be selected in the project.

Click on **Project: Change OS Version** in the main menu.

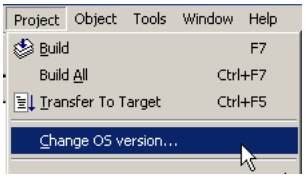


Fig. 49: Changing the opterating Sytem Version in AS 2.x

Click on **Project: Change Runtime Version** in the main menu.

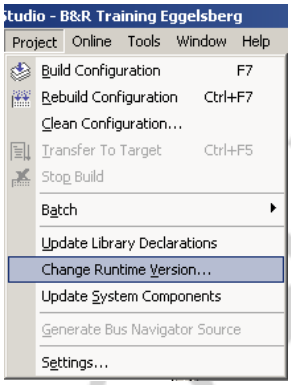


Fig. 50: Changing the operating System Version in AS 3.x

You receive a message that the project should be recompiled after the change because the library versions can also change in a new operating system.

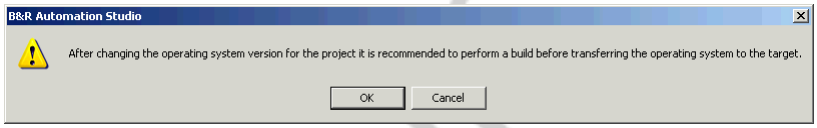


Fig. 51 Message

Use the combo box to select the desired operating system and confirm your selection by clicking on OK.

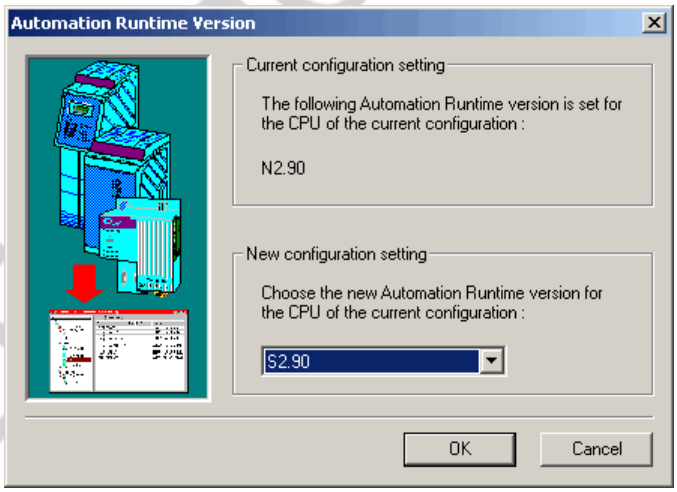


Fig. 52 Selecting the new operating system version

Compile your project via **Project:Build All**.

7.3 Online connection

Follow the steps below to establish an online connection.

Select the **Tools**: **Options** menu item in Automation Studio.

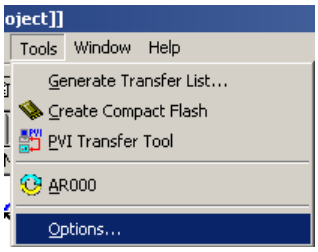


Fig. 53 Opening the connection settings in AS 2.x

Select the **Online**:**Settings** menu item in Automation Studio.

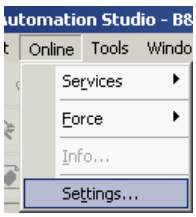


Fig. 54 Opening the connection settings in AS 3.x

Select an existing serial connection in the combo box or add a new one with the **Add...** button.

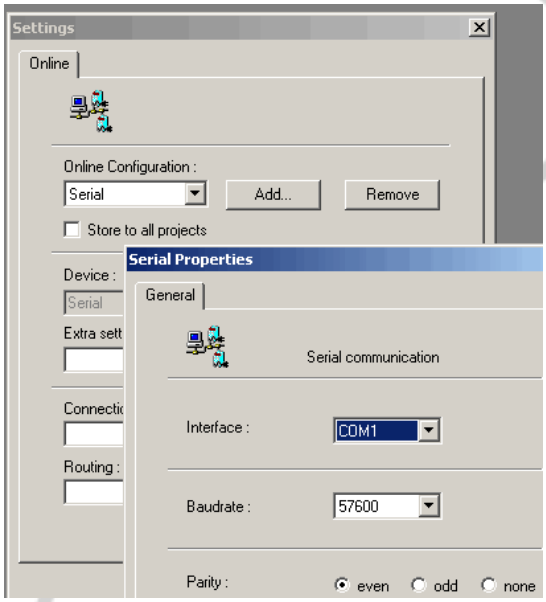


Fig. 55 Checking the settings

The status bar will indicate whether there is now a connection to the target system.

Make sure that the correct interface on the PC is specified.

The status bar will indicate whether there is now a connection to the target system.

7.4 Downloading the operating system

Before downloading, make sure the correct configuration is active.

7.4.1 Transferring with Automation Studio

Follow the steps below to transfer an operating system to the target system with Automation Studio:

Select **Project:Services:Transfer Operating System...** from the main menu.

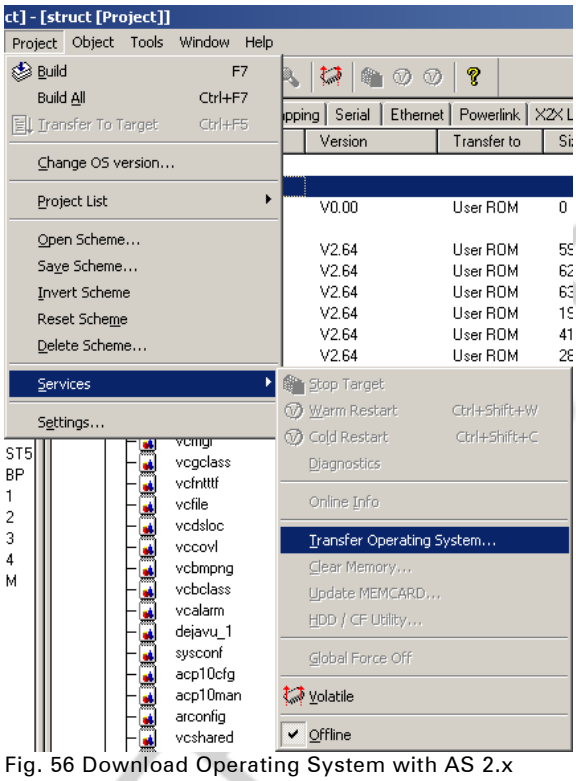


Fig. 56 Download Operating System with AS 2.x

Select **Online:Services:Transfer Operating System...** from the main menu.

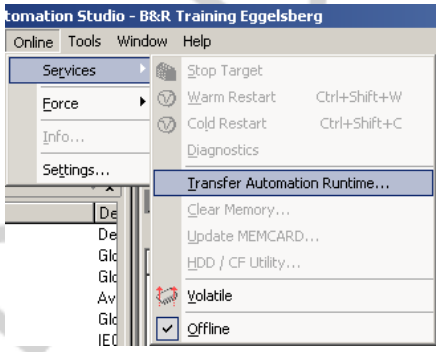


Fig. 57: Download Operating System with AS 3.x

The operating system can be selected in the next dialog box. The operating system configured in the project is always recommended.

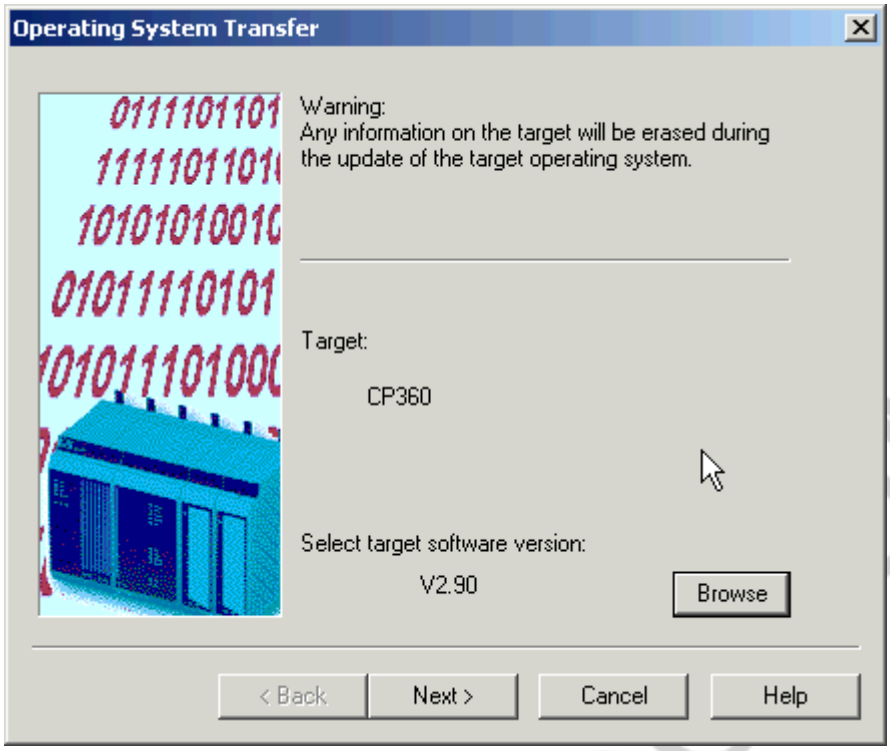


Fig. 58 Selecting an operating system version

After selecting **Next**, you can select whether modules that have been set to be transferred to System ROM in the software configuration should be transferred along with the operating system. For example, this makes sense when updating via Ethernet. When doing this, system ROM must be set as the target memory in the system configuration (sysconf). This ensures that the Ethernet settings will be present after the operating system update.

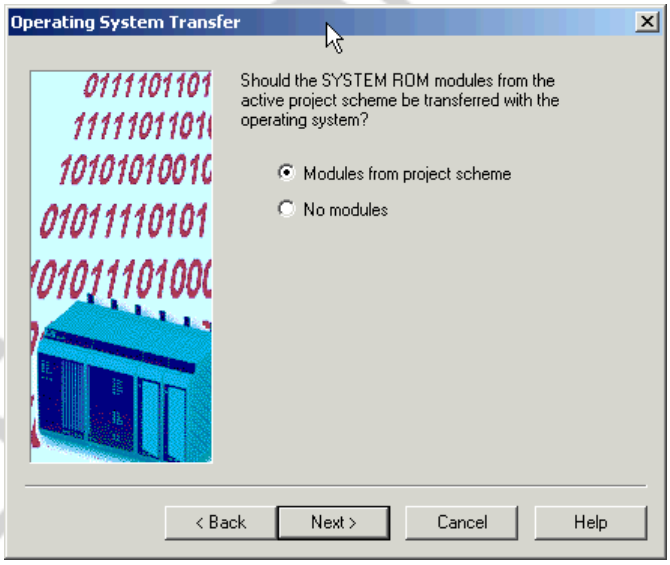


Fig. 59 Transferring project modules

Selecting **Next** then shown an overview of the selected operating system and the target hardware.

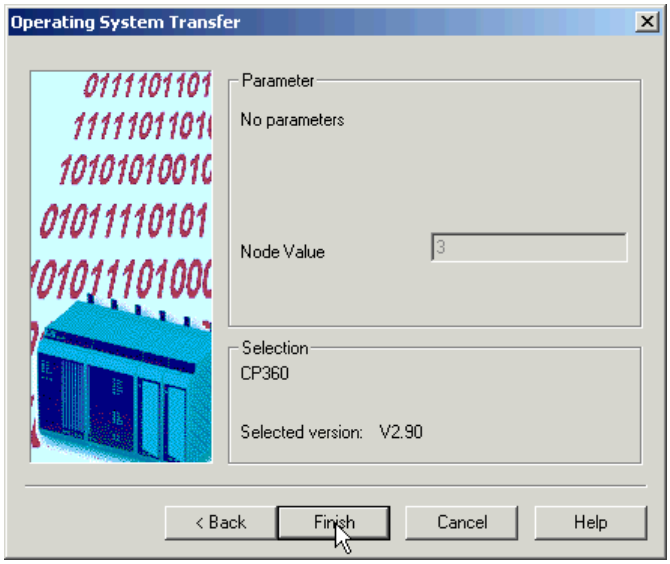


Fig. 60 Summary

Clicking on **Next** erases the flash memory.

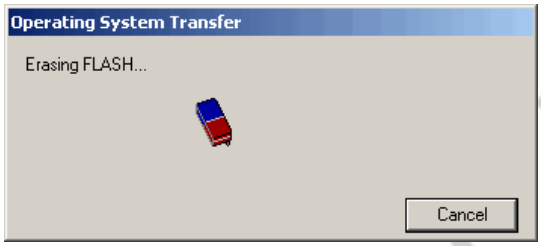


Fig. 61 Erasing memory

Once flash memory has been erased, the operating system starts downloading.

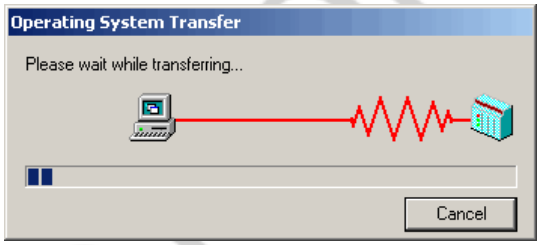


Fig. 62 Transferring the operating system

Follow any additional instructions from Automation Studio.

Note:

A download can be interrupted at any time by clicking on **Cancel**. The previous version of the operating system on the target system continues to be used.

7.4.2 Downloading with PVI Transfer

PVI Transfer can be used for diagnostic and downloading purposes. It's also possible to use CompactFlash directly on the PC, providing another alternative for downloading the operating system.

PVI Transfer works with transfer lists that are executed sequentially.

The CompactFlash card needs to be inserted into a suitable CompactFlash reader or adapter connected to the PC.

Select the **Tools: Generate Transfer List...** item from the main menu.

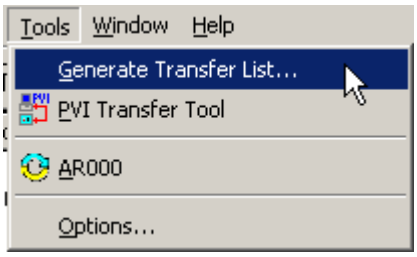


Fig. 63 Creating a transfer list

Check the options **Generate complete transfer list** and **Include operating system**.

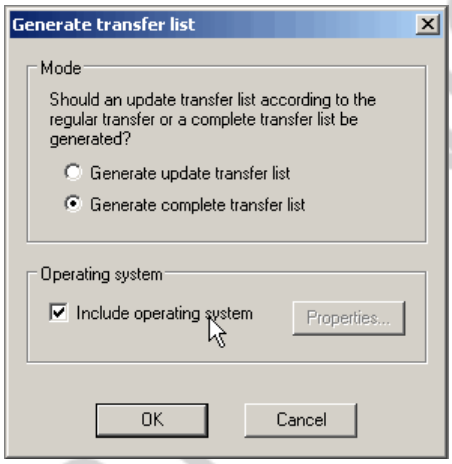


Fig. 64 Including the operating system

The transfer list is then created.

```
* Transferring
* Transferring
* Transferlist Y:\
* End transfer list
```

Fig. 65: Transfer list created

Select the **Tools: PVI Transfer Tool** menu item.

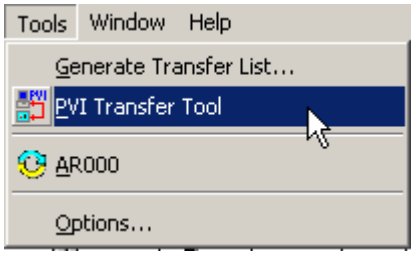


Fig. 66 Starting PVI Transfer

The correct transfer list has already been loaded automatically. In the PVI Transfer tool, select the **Tools: Generate Compact Flash...** menu item.

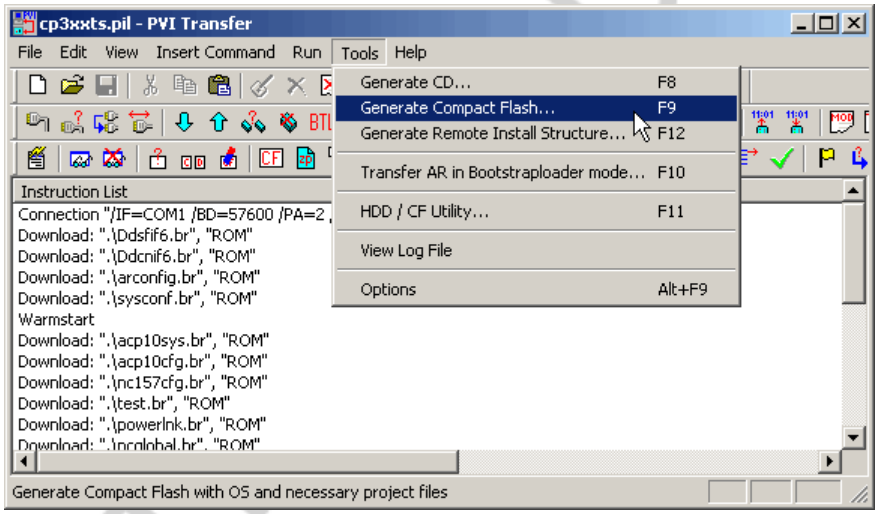


Fig. 67 Generating CompactFlash

Select where the CompactFlash card is located with **Select Disk...**.

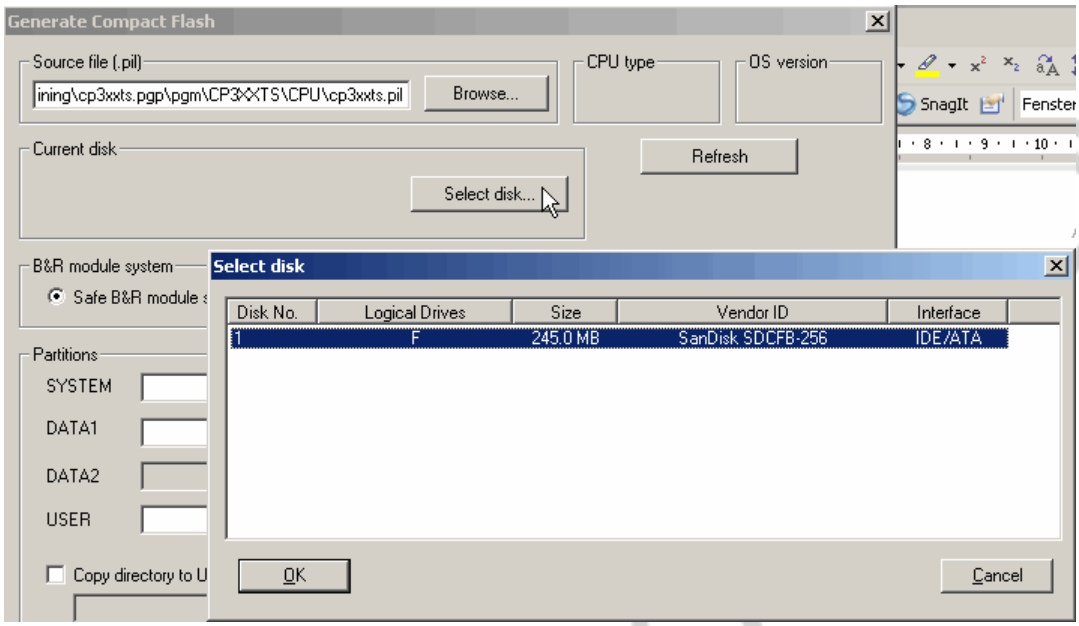


Fig. 68 Selecting the disk

Use the **Generate Disk** button to begin creating the CompactFlash.

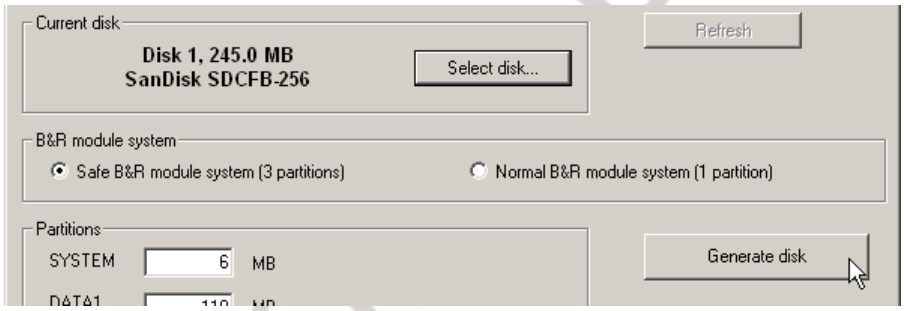


Fig. 69 Generating the disk

Follow any additional instructions.
Insert the Compact Flash that you have set up into the CPU and turn it on.

Note:

The Compact Flash card should always be removed using the Windows function "Safely Remove Hardware".

Summary

The operating system serves as the interface between the user and the hardware. In addition, it's responsible for resource management, especially with regard to time and memory.
Multitasking enables a modular structure. The distribution of the application into tasks in connection with different task classes enables optimum utilization of the resources.

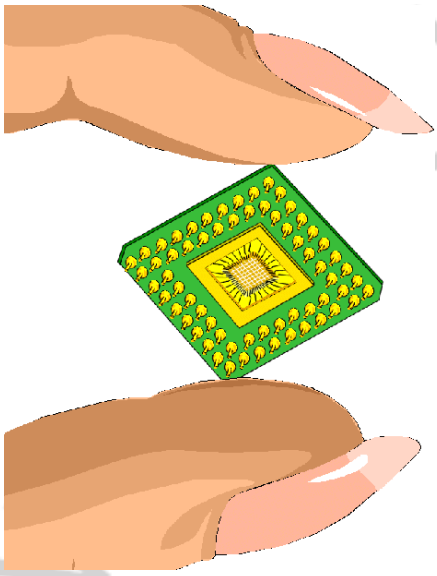


Fig. 70 Microchip

Adapting the application to the multitasking system keeps all aspects of timing in check. Different software parts can therefore be prioritized as needed. Parts of the application that need to be executed quickly and often run in a faster task class that those that, while certainly not unimportant, may not be as time-critical. This makes it possible to optimally configure the performance of the application and machine while using the existing resources selectively.

Notes

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

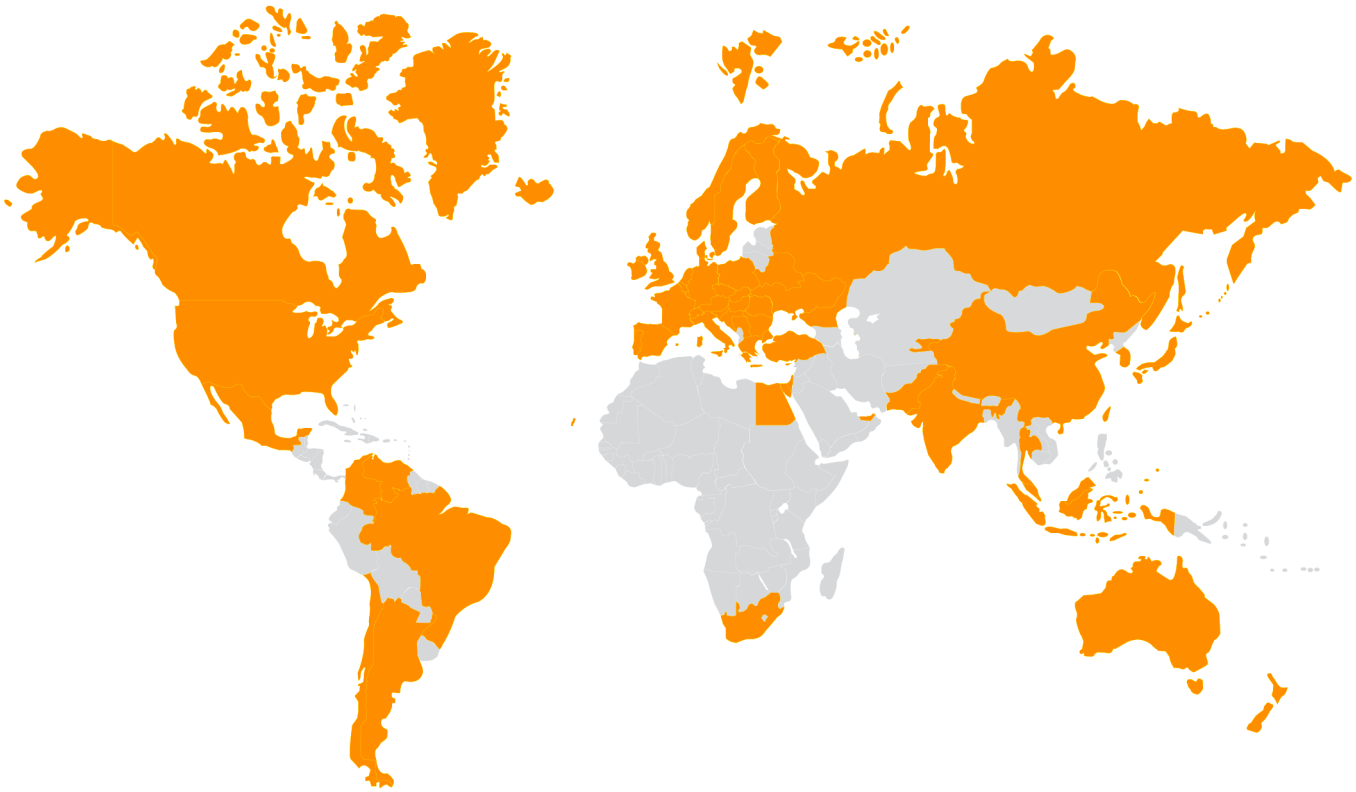
**) see Product Catalog

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

TM213TRE.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