Jetter-Conform Programming

Introduction

When programming with SYMPAS, some special features need to be taken into account with respect to the behavior of the operating system of the controller.
Furthermore, some guidelines should be observed when designing and generating an application program.

This application note gives an overview of special features and guidelines, thus enabling the programmer to realize fast, efficient, clear, and easily maintainable programs.

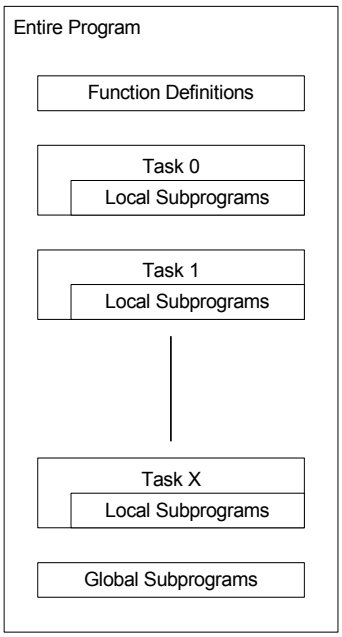
Essential Information about Programming in SYMPAS

Unlike conventional controllers, the programs of control systems by Jetter AG are not processed cyclically.

Therefore, the response time does not depend on the length of the program since only the input conditions required to continue the control flow are checked permanently (all others are not checked).

Program Structure

Basic Program Structure



- Function definitions, if existing, have to be placed before task 0. These functions can be called from any task and any subprogram.
- The entire program can comprise up to 32 tasks.
- The first task is always task 0. The subsequent tasks can be numbered in any order. It makes sense, however, to number the tasks in their correct order to enhance clarity and to save controller memory for the task management. This is important when many function call levels are realized.
- Subprograms directly following a task are referred to as local subprograms. These local subprograms can only be called from this task.
- Subprograms following the last task are referred to as global subprograms. These global subprograms can be called from any task.

Jetter AG
Gräterstrasse 2
D-71642 Ludwigsburg
Germany

Phone - Switchboard: +49 7141 2550-0
Phone - Sales: +49 7141 2550-530
Phone – Technical Hotline: +49 7141 2550-444

Telefax: +49 7141 2550-425
E-Mail - Sales: sales@jetter.de
E-Mail - Technical Hotline: hotline@jetter.de
Internet address: http://www.jetter.de

Functions

- Functions can call local subprograms. However, parameters and variables of the function cannot be accessed from this local subprogram.
- Parameters and variables of functions do not occupy any user register in the controller. They are directly managed by the operating system of the controller and are created in the controller memory during program runtime as soon as the function is called. After completed function, this storage position is deallocated again.
- Functions can be called from any task. For each function, up to 10 transfer parameters and up to 10 internal variables can be defined. Besides, a return value of the function is available.
- Calling the same function several times from different tasks does not lead to an undefined state of the application program.

Note

Important for troubleshooting:

The content of parameters and variables can only be read in the setup screen when the program flow is present in this function.

If this function is inactive, i.e. is currently not being processed by the operating system of the controller, then its parameters and variables are not stored in the memory.

Subprograms

- Symbol names can be assigned instead of numbers. The symbol "**!**" is assigned automatically and need not be replaced by a value.
- There are **local** and **global** subprograms. Global subprograms follow the last TASK. They must have a number as parameter ("!" does not work). Global subprograms can be called from any task. Local subprograms can only be called by the tasks under which they are located.
- If subprograms are used in functions, it is not possible to transfer parameters and define variables. User registers must be used to transfer values as well as for internal intermediate results.
- Like functions, global subprograms can be called from any task. However, calling the same subprogram several times may modify intermediate results of previous subprogram calls that have not been processed yet. This can lead to errors.

Tasks

Essential Information on Tasks

Each task must constitute a complete unit. The last instruction of each task must be an unconditional jump to a label in the same task, or to the beginning of the task. It is not possible to jump from one task to another or to call a local subprogram of another task.

Note

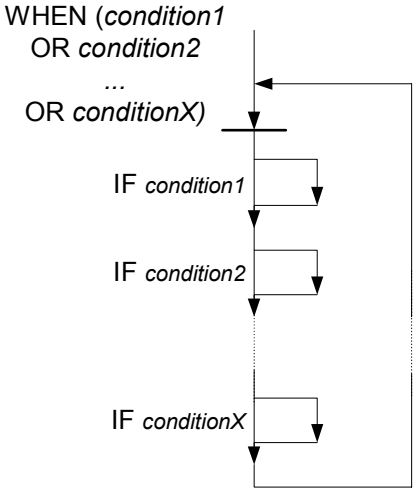
The duration of program execution primarily depends on the number of tasks being used. The program length is only of secondary importance to the processing time. To allow fast processing of a program, clever programming and thus a limited number of tasks is crucial.

Waiting Task

Tasks should not be programmed as continuous tasks. A continuous task is a task which, for example, only consists of a sequence of IF instructions. For this task, the controller needs permanent computing time although the program not necessarily carries out an action.

There should not be more than two permanently running tasks in the application program. These are

- the monitoring task (checking the emergency stop function, overtemperature, control voltage, operating pressure of the compressed air, …),
- the communication task (sending and fetching registers via N-SEND and N-GET instructions).



If the task has to wait for events (e.g. input signal is present, a certain register value or flag state has been reached, or a key on the user interface was pressed), then the task should be stopped by a WHEN instruction until the event occurs.

Next, branching to the individual event handling can be carried out by IF/THEN queries.

For illustration: you do not regularly pick up the phone just to check whether someone is calling. Instead, you wait until the phone rings before picking up the phone.

Few Tasks

When creating the program, you should consider carefully whether it is necessary to realize an intended function in a separate task.

Each additional task means additional handling effort (time for the task switch) for the operating system of the controller which lengthens the runtime of the program.

Special Tasks

Some tasks should be realized in a separate task:

- Communication: communication via JetWay, etc., should be performed by one single task only. (In this case, network instructions can be made interruptible.)
- User interface: if several tasks access one user interface it might occur that, for example, displayed text that should be readable for the user is overwritten by other parts of the program.
- Error handling: it should be possible to respond to faults in the system quickly, irrespective of the condition of the application program.
- Accessing axes: several tasks should never access one axis since this might result in uncontrolled motion of the axis.

Task Switch

Starting with task 0, switching from one task to another is always performed after a fulfilled task switching condition.

Example for a program with 4 defined tasks:

Task 0	Task 1	Task 2	Task 3	Task 4	Task 0	Task 1	Task 2	Task 3	Task 4	Task 0	Task 1	Task 2	…

A task switch can take place under the following conditions:

- Immediately after a network instruction
- At each DELAY (also when the time is unequal 0) or USER_INPUT instruction
- In case of an unfulfilled WHEN condition
- After expired task switch timeout and reaching a THEN instruction
- In case of a GO TO instruction
- In case of an unfulfilled IF condition

Some of the above mentioned task-switch conditions can be configured by using special registers of the operating system of the controller.

Prioritized Task

It is possible to prioritize a user task by specifying the task number in a special register. This task is then executed after each completed user task.

Exemplary order with task 1 as priority task:

Task 0	**Task 1**	Task 2	**Task 1**	Task 3	**Task 1**	Task 4	**Task 1**	Task 0	**Task 1**	Task 2	**Task 1**	Task 3	…

It is also possible to prioritize network tasks and LCD tasks. (For more details, please refer to the manual of the respective controller.)

GO TO

Instead of the program sequence

```
IF rMain = 2
THEN
GO TO...
IF rMain = 3
THEN
GO TO ...
```

use the instruction

```
GO TO R(rMain)
```

and add the relevant jump labels to the subprograms.

Important

- All labels to be reached by the GO TO instruction must exist. Therefore we recommend to assign the labels in unbroken order.
- It is advisable to insert a LIMITS instruction prior to the GO TO instruction: LIMITS [Reg=rMain, lower=rLowerLimit, upper=rUpperLimit]

Communication

Minimum Data Traffic

Unnecessary data traffic lengthens the program runtime and causes additional load on the communications line which might block other nodes.

Unnecessary data traffic can be avoided by

- only transmitting data that has actually changed (e.g. only send outputs to the remote I/O if their condition has changed)
- by transmitting as much data as possible per access (use overlaid registers for inputs/outputs, flags)
- processing data locally, if possible. Only data that is really needed by the remote station should be transmitted.

Symbolic Programming

The numeric parameters of the programming language can be replaced by symbolic names. This considerably enhances readability and comprehensibility of the program source text.

Self-explanatory designations should be used for the symbol names.

Additional effect: if the program needs to be modified, e.g. because an output terminal is reassigned, then this modification can easily be realized by using symbolic programming.

It is useful to agree on some rules for choosing symbol names. For instance, the initial letters of the symbol names for inputs, outputs, flags, registers, etc., should be defined uniformly.
This has two considerable advantages:

1. Enhanced readability of the program source text since each symbol name not only indicates its meaning/function (by the self-explanatory name) but also its type (inputs, outputs, flags, registers, etc.).

2. When generating the program, it is possible to display, for example, all inputs by entering *i?* .

We recommend to use the following initial letters for symbol names:

- **o** ... Outputs
    - **ov** ... virtual outputs to configure expansion modules (e.g. N-IA4)
- **i** ... Inputs
- **f** ... Flags
    - **fs** ... Special flag

- **r** ... Registers
    - **rAx** ... Local axis registers
    - **rs** ... Special registers on the CPU
        - **rsV** ... Special register for Viadukt
    - **rf** ... Floating-point registers
    - **rm** ... Registers on the expansion modules
        - **rmAx** ... Axis registers on expansion modules
    - **rp** ... Registers that are used as pointers
    - **rt** ... Registers that contain text (text registers)
    - **rt** ... Registers where a time expires
- **l** ... Label
    - **ls** ... Label for a subprogram
    - **lm** ... Label for program parts that handle an expired WHEN_MAX
    - **le** ... Label for program parts that handle error conditions
- **t** ... Task
- **n** ... Numbers
    - **nb** ... Numbers for individual bits within a register
    - **ne** ... Numbers used as error code
    - **ni** ... Numbers used as current constants (e.g. for N-SM1D)
    - **nc** ... Numbers used as commands (e.g. for axes)
    - **no** ... Numbers used as offset (e.g. for axes or counters)
    - **nt** ... Numbers used as time constants
    - **nv** ... Numbers used as Viadukt masks