Final Project

Development and Implementation of the **R**ealistic **R**obot **S**imulation Interface for the Volkswagen Robot Controller VRS1

Göksel DEDEOĞLU

August 1995 - January 1996
Volkswagen AG,Wolfsburg

Advisor: Prof. Seta BOĞOSYAN
Istanbul Technical University
Electrical and Electronic Faculty
Control & Computer Engineering Department

Supervised by :

Prof. Dr. -Ing. J. Hesselbach
Dr.-Ing. Kerle
Dipl.-Ing. R. Thoben
Technische Universität Braunschweig
Institut für Fertigungsautomatisierung
und Handhabungstechnik

Dipl.-Ing. P. Beske
Volkswagen AG
Elektroplanung Abt.
Wolfsburg

Table of Contents

Introduction

The aim of the Realistic Robot Simulation (RRS) Project has been to define a neutral software interface which would allow Computer Aided Robotics Tools to use original robot controller algorithms for a more accurate simulation. Derived from the original controller software, Robot Controller Simulation (RCS-) Modules can be supplied by controller manufacturers as black-boxes to any simulation system supporting the RRS-Interface.

The objective of this work is to develop the RCS-Module for the Volkswagen Robot Controller VRS1 and to integrate it into the ROBCAD system of Tecnomatix Technologies. As such, this has been the very first RRS-Software package developed by the Volkswagen Group, which has been among the project's partners since its start in 1992.

Design approach

In order to ease future maintenance efforts for the RCS-Software, particular emphasis has been put on keeping the original controller software parts of the module as intact as possible. Consequently, the first step has been to implement an interprocess communication library under the UNIX operating system. As a result, the original path module has become fully portable on UNIX compatible platforms.

The second step consisted of developing the RRS-Services which could be supported by the original controller. During this phase, a shell-program has been used for testing the RCS-Module, making the desired RRS calls instead of the CAR-Tool.

Finally, the module has been integrated into ROBCAD running on a *Silicon Graphics-Indigo* platform by using the *Controller Modelling Language* technique of this CAD system.

Results

After six months of development which covered the time period from August 1995 until January 1996, a working RCS-Module has been achieved and put into operation under ROBCAD. The RRS-Interface for VRS1 has already proven to be of considerable benefit, since path deviations up to 90 milimeters during PTP-motions could be easily observed in a number of cases.

Unfortunately, ROBCAD's limited RRS-Interface capabilities did not allow the RCSVW-Module to be tested to the desired extend before a possible industrial use. Moreover, the tight time frame in which the RCSVW-Module has been developed, implemented and integrated into ROBCAD did not allow a sound investigation of the sources of deviations between ROBCAD's motion planner and the original controller software. For this reason, in most of the available examples, graphical methods have been used to depict these differences.

1. The Realistic Robot Simulation Project

1.1 The purpose of the RRS Project and its benefits

Various interactive and graphics-based tools have been introduced in industry to enable computer aided planning, simulation and off-line programming of industrial robots. These tools usually provide visualizations of robotized production cells, based on geometrical and kinematic modelling, as well as models for the behaviour of the controllers involved. Until recently, this modelling was mostly based on default controller models, whose replacement with the original one usually required extensive measurements of the robot behaviour in reality. Nevertheless, these efforts did not lead to sufficient simulation accuracy for down-loading programs and executing them without reteaching.

In order to improve the situation, the companies from the European Automotive Industry initiated the project Realistic Robot Simulation (RRS), in which suppliers of robot controllers and Computer Aided Robotics (CAR-) Tools cooperate in order to define a common interface for controller simulating software. As technical experts they involved manufacturers of robots and robot controllers (ABB, Comau, Fanuc, Kuka, Renault Automation, Siemens, Volkswagen), and manufacturers of off-line programming systems (Dassault Systeme, Deneb, Silma, Tecnomatix). For neutral project management, the Fraunhofer-Institute for Production Systems and Design Technology (IPK Berlin) was selected as an independant research institute.

Benefits of Robot Controller Suppliers

By implementing RRS-Interfaces for their software, controller suppliers can provide simulation products which assure a better utilization of CAR-Tools at their customer's site. These simulation softwares can be used in all CAR-Tools supporting this interface. Hence, controller suppliers do not need to implement different simulation products for every CAR-Tool. Furthermore, they can focus on the development of dedicated products without reimplementing the general parts of CAR-Tools. Consequently, this effort allows the controller supplier to minimize implementation efforts.

Benefits of CAR-Tool Suppliers

Suppliers of CAR-Tools do no longer have to implement and verify specific controller models for accurate simulation of their CAR-Tools. Since the original controller software will be used, verification will become obsolete. Up to now, CAR-Tool suppliers have implemented and verified their controller simulation models for each controller type. Therefore, this approach saves development resources considerably.

Benefits of CAR-Users

By using original controller software within CAR-Tools, the simulation accuracy of the industrially applied CAR-Tools will be improved, which will effectively reduce down-times of the manufacturing equipment. Once a new controller type is acquired, the

automotive companies can also buy the corresponding simulation product for a precise simulation. Without a long implementation and verification phase, a precise simulation can be used during the initial operation phase of a new controller type.

Realization of the project

The RRS-Project was started in January 1992 by defining the requirements of the interface. As the participants agreed to concentrate on the controller software for motion behaviour and kinematics, simulation deviations from reality of less than 0.001 radians for planned joint values and less than 3% for cycle times have been desired. It was required to have several controllers of the same or different controller type and manufacturer running in parallel in the same simulation. Finally, the interface had to support data consistency of simulated and real controllers.

From March '92 on, the first sketch of the interface has been elaborated. The sketch has been continously enriched and improved by feasability studies and by cross checking with the functionalities of the involved controllers. After more than 32 project meetings version 0.0 of the RRS-Interface Specification was available in December '92.

Implementation work on first Robot Control Simulation (RCS-) Modules began in January '93. Until May '93 RCS-Modules with a dummy functionality have been implemented and implementations including original controller software have been distributed to the manufacturers of off-line programming systems in July '93. In December '93, tested integrations of five RCS-Modules in four simulation systems and the verified version 1.0 of the RRS-Interface Specification have been presented.

1991	July	Formation of the project
1992	January	Definition and review of requirements
	March	First sketches of architecture and services
	November	Development of calling conventions and test software
	December	Completion of version 0.0 of the RRS-Interface Specification
1993	January	Implementation of shell version with dummy functionality
	May	Implementation of principal services
	July	Distribution of object code
	September	First complete prototypes
	December	Presentation of results with first working prototypes
1994	January	Final review of version 1.0 of the RRS-Interface Specification
1995	August	Start for the development of Volkswagen's RCS-Module
	September	Version 1.1 of the RRS-Interface Specification
1995	November	Integration work of the RCSVW-Module with ROBCAD
1996	January	First RCSVW test motions under ROBCAD

Table 1.1 : Development of the RRS-Project

1.2 Off-line programming and simulation with Computer-Aided-Robotics Tools

1.2.1 Off-line programming

An off-line programming (OLP) system can be defined as a robot programming language which has been sufficiently extended, so that the development of robot programs can take place without access to the robot itself but by means of computer graphics /CRAxx/. Off-line programming systems are important both as aids in programming industrial automation as well as platforms involved in robotics research.

The use of off-line programming and simulation systems to program industrial robots enables the shift of program creation and optimization away from production, thereby reducing down times in production cells. As such, off-line programming also offers the possibility of using simulation technology to experiment with a variety of scenarios and processes, in order to optimize processes without interfering with production or endangering the cells. Even before a unit is constructed and put into operation, a variety of errors and weak points can be identified and avoided.

The most important advantages that OLP systems offer may be summarized as follows :

- Possibility of programming during production
  => Increase in availability of production equipment

- Dramatical reduction of on-line programming time in robotized cells
  => Set-up periods are shortened

- Programming in parallel with the construction of equipment
  => Run-up times are reduced

- Determination and optimization of cycle times

- Reachability tests and investigations of collision with the equipment or other robots

- Additional safety during the on-line programming

Today, a great deal of time and expertise is required to install a robot in particular application and bring the system to production readiness. There are a number of factors that make robot programming a difficult task. Programming robots, or any programmable machine, has particular problems which make the development of production-ready software even more difficult, most of which arise from the fact that a robot manipulator interacts with its physical environment. Even simple programming systems maintain a world model of this physical environment in the form of locations of objects and have knowledge about presence and absence of various objects encoded in program strategies.

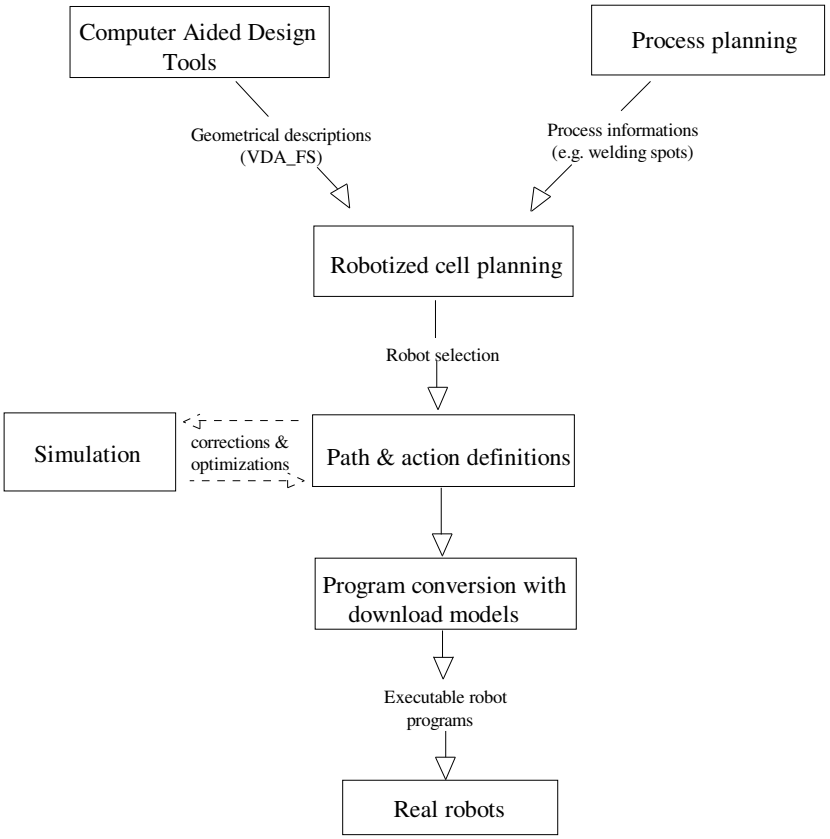


Figure 1.1 : Off-line programming

During development of a robot program, it is necessary to keep the internal model maintained by the programming system in correspondance with the actual state of the robot's environment. Interactive debugging of programs with a manipulator requires frequent manual resetting of the state of the robot's environment. Such state resetting becomes especially difficult when the robot performs an irreversible operation on one or more parts.

1.2.2 Simulation

The ideal simulation approaches reality so closely that programs developed off-line can be loaded onto and executed by real controllers without having to correct them at the shop floor. Although the current technological state does not allow such a precision, simulators have already proven to be of economic benefit /BER94/.

In contrast to teach-in programming of a robot which requires basically a high accuracy in terms of repeatability, for off-line programming the accuracy in the positioning of the robot plays the dominant role /BERN94/. Absolute positioning accuracy depends on the quality of the manufactured robot and the accuracy of the robot model used for motion control. To ensure quality manufacturing and to identify robot model parameters accurately, advanced measuring procedures and model-based parameter identification methods are required. These procedures and methods make up the techniques called

robot calibration. Calibration results are a set of identified robot parameters which can be used by the robot manufacturer as a check on the quality of robot production and by the robot user to improve the robot's absolute positioning accuracy, using for instance these data for the compensation of off-line generated robot programs.

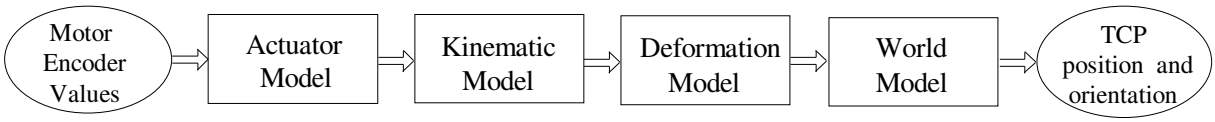


Figure 1.2 : Robot model /BERN94/

As illustrated in the figure 1.2, the robot model is defined as an integration of four models; the actuator model, defining the mechanical relationship between robot motors and joints; the kinematic model, describing the robot's overall movement; the deformation model, characterising the compliance in the robot joints and links; and the measurement target model, specifying the tool-center-point (TCP) with respect to the robot flange.

The absolute pose accuracy of the robot is today restricted by rough kinematic modelling. By *robot calibration*, it can be improved close to repeatability. Research here includes development of suitable models, low cost measuring equipment, data acquisition and management systems and automatic generation of appropriate calibration data sets.

A central cause for deviations is the motion behaviour of the robot controller. Its simulation is still unsatisfactory, despite enormous efforts that have been undertaken to rebuild the interpolation of real controllers in simulation. Figure 1.3 shows an example of deviations between the original robot controller's (RCSVW) and a simulated one's (ROBCAD) behaviour.

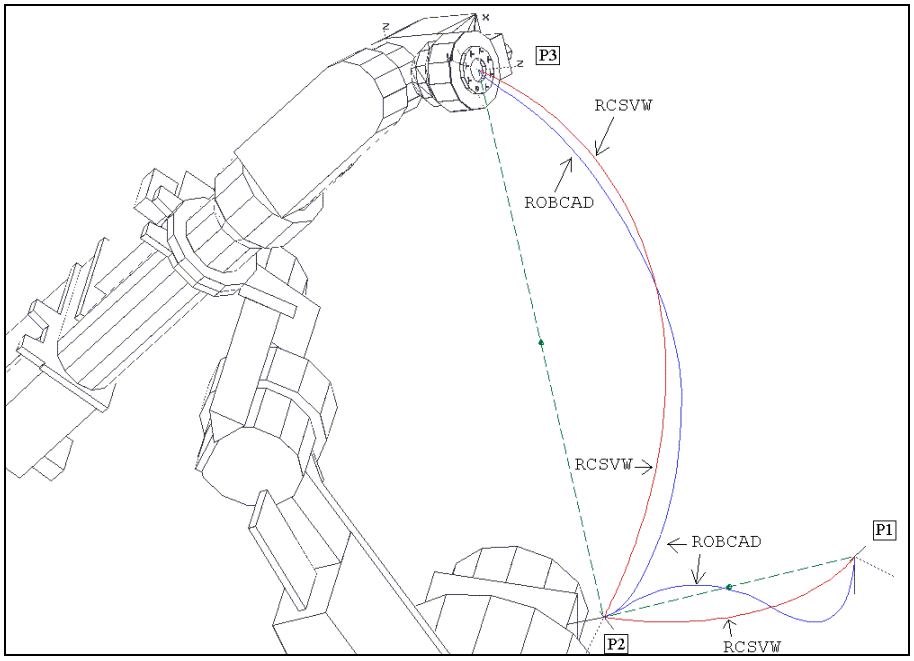


Figure 1.3 : Deviations between two different robot controllers

1.2.3 Deviation of Simulation and Reality in Robotics

The nature of simulation implies a certain degree of abstraction from reality /BER95/. For accurate and realistic simulation of a technical system, each component of the system has to be modeled in a adequate way. With respect to practical application the relevance of each component for the simulation purpose as well as the effort required for modelling have to be considered.

Figure 1.4 illustrates the major components for an industrial robot involved in motion programming. User interface, language system, path control, transformation and parts of the servo control are mainly software components. Parts of the servo control, power amplifier, position measurement and motors are mainly electronic or electromechanical components, while gears and mechanical structure are pure mechanic components.

Robot controllers of different manufacturers provide different language systems that are tailored to the specific capabilities of the controllers like interpolation procedures and condition handlers. On the other hand, most simulation systems provide a general language that is also used in task specific programming environments. The programs are translated to the specific native controller language and down-loaded afterwards. In addition, several simulation systems provide also emulations of native controller languages. Programs that are up-loaded from the real controller may be executed there.

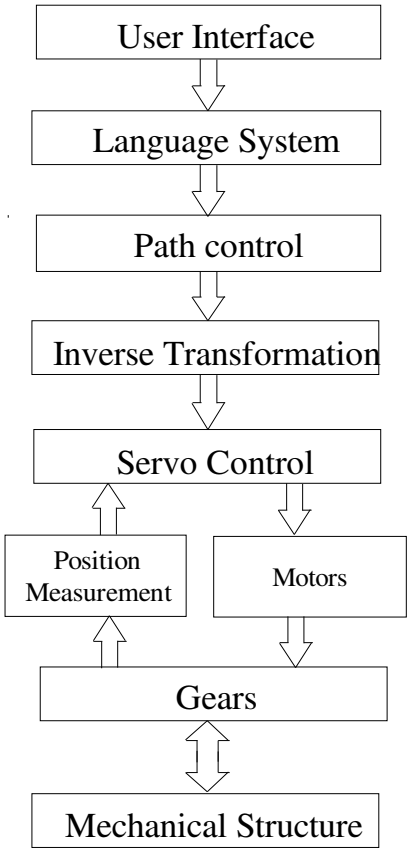


Figure 1.4 : Major robot components involved in robot programming /BER95/

Several instructions like *ptp* or *linear motion* are common to most controllers. But some other language elements like *fly-by* have not only a different syntactical structure, but also different underlying semantics. For instance, fly-by may be commanded and executed with respect to a fly-by zone, to a speed reduction or to a precision sphere. Providing a superset of all mechanisms would enable the programmer to specify semantics that may not be executed by a real controller. Providing no fly-by mechanism would make the universal language completely inefficient. The problem is principally avoided by emulating the native language. However, other problems arise : on the one hand, it requires a considerable effort since the language systems of several controllers have to be rebuilt for several simulators. On the other hand, rebuilding a system is error-tedious.

The path control of robot controllers provides the widely used motion types as ptp, linear and circular and often special motion types like weaving, spline motions or optimized motions. Additionally, features like fly-by, event generation and conveyor tracking are provided. The corresponding mechanisms may be modified by a variety of parameters like speed and orientation follow-up.

In order to move simulated robot arms, most simulation systems also provide a path control system. The path controls of robot controllers and simulation systems provide similar functionality and parametrization. However, they differ obviously in the extend of functionality and its performance. Such deviations arise from the orientation follow-up and fly-by during linear and circular motion. In addition to the inaccuracies in space, also the execution times of motions deviate. This leads to considerable errors in cycle-time simulation.

Several attemps have been made for improvement of the simulation accuracy for path control: Special parametrizable path controls that include a variety of interpolation procedures with a number of combinations have been developed, the simulation systems have been extended for open interfaces that allow the integration of path controls, special interfaces have been designed for the integration of path controls of specific real controllers into simulation systems.

The inverse transformation of a robot controller converts cartesian set points on paths to the corresponding joint values of the robot. Since the transformation is computationally expensive and has to be performed in high frequency, the transformation algorithms are simplified and do not precisely match the real kinematics of the mechanical arm. This causes absolute deviations between programmed and executed paths.

The servo control, motors, gears and position measurement for the control loop that keeps the real robot arm on the path given by the path control. Dependent on path geometry, speed, acceleration, etc., the mechanical structure follows the path more or less precisely. In simulation systems, dynamic models of these components are not in common use. One reason lies in the fact that reliable values for required parameters like friction are difficult to obtain. Another reason lies in the more important deviations caused by the path control.

1.3 RRS-Interface Specifications

Using the Realistic Robot Simulation Interface, CAR Tools have access to the original controller software by means of RRS-Services. These can be categorized as follows :

Base services

INITIALIZE
RESET
TERMINATE
GET_ROBOT_STAMP
GET_RCS_DATA
MODIFY_RCS_DATA
SAVE & LOAD RCS_DATA

Kinematic and conversion services

GET_INVERSE_KINEMATIC
GET_FORWARD_KINEMATIC
MATRIX_TO_CONTROLLER_POSITION
CONTROLLER_POSITION_TO_MATRIX
GET_CELL_FRAME
MODIFY_CELL_FRAME
SELECT_WORK_FRAMES

Principal motion services

SET_INITIAL_POSITION
SET_NEXT_TARGET
GET_NEXT_STEP
SET_INTERPOLATION_TIME

Motion modification services

SELECT_MOTION_TYPE
SELECT_TARGET_TYPE
SELECT_TRAJECTORY_MODE
SELECT_ORIENT_INTERPOL_MODE
SELECT_DOMINANT_INTERPOL
SET_ADVANCE_MOTION
SET_MOTION_FILTER
SET_OVERRIDE_POSITION
REVERSE_MOTION

Motion parameter services

SET_JOINT_SPEEDS
SET_CARTESIAN_POSITION_SPEED
SET_JOINT_ACCELERATIONS
SET_CARTESIAN_POSITION_ACCEL.
SET_CARTESIAN_ORIENTATION_ACCEL
SET_JOINT_JERKS
SET_OVERRIDE_SPEED
SET_OVERRIDE_ACCELERATION

Fly-by and point accuracy services

SELECT_FLYBY_MODE
SET_FLYBY_CRITERIA_PARAMETER
SELECT_FLYBY_CRITERIA
CANCEL_FLYBY_CRITERIA
SELECT_POINT_ACCURACY
SET_POINT_ACCURACY_PARAMETER
GET_CURRENT_TARGETID

Tracking services

SELECT_TRACKING
SET_CONVEYOR_POSITION

Condition handling services

DEFINE_EVENT
GET_EVENT
STOP_MOTION
CONTINUE_MOTION
CANCEL_MOTION

Weaving services

SELECT_WEAVING_MODE
SELECT_WEAVING_GROUP
SET_WEAVING_GROUP_PARAMETER

1.3.1 Overview of the functionality

CAR-Tools initialize an instance of robot controller model using the service INITIALIZE /RRS95/. This service returns the parameter *RCSHandle*, whose value identifies uniquely this robot instance and which has to be passed to each service call working on it. In case the service INITIALIZE is successful, the model is considered to be valid, thus all RRS-Interface services may be applied to it. At this point, GET_ROBOT_STAMP may be called to get robot's signature data.

For jogging or passing start positions, the service SET_INITIAL_POSITION may be applied. After calling this service, SET_NEXT_TARGET can be used to pass the target position for the first move.

The key service for motion reporting is GET_NEXT_STEP. After each call, it reports a status which gives important information for further possible calls : If the status is 0, GET_NEXT_STEP returns the next interpolated position. By succesive calls of this service, motions can be scanned with the highest resolution. If this service returns 1, this means that the controller needs a new target position. In this case, a new target position is passed by the service SET_NEXT_TARGET.

The desired type of motion may be specified by SELECT_MOTION_TYPE, SELECT_ TRAJECTORY_MODE, SELECT_ORIENTATION_INTERPOLATION_MODE and SELECT_DOMINANT_ INTERPOLATION. A number of SET- commands are available for the modification of the speed profiles. Fly-by behaviour and point accuracy are determined by the services SELECT_FLYBY_MODE, SET_FLYBY_CRITERIA_ PARAMETER, SELECT_FLYBY_CRITERIA, CANCEL_FLYBY_CRITERIA, SELECT_ POINT_ ACCURACY and SET_ POINT_ ACCURACY_PARAMETER.

To change the look ahead of the interpolation, SET_ADVANCE_MOTION may be called. REVERSE_MOTION commands the controller to move backwards on a path. SET_ OVERRIDE_SPEED changes the speed and SET_OVERRIDE_ POSITION adds offsets to the current TCP while the robot moves.

GET_NEXT_STEP returns 2 if the last given target is reached or if the speed is zero because of a STOP_MOTION command. After STOP_MOTION, moves may be continued by CONTINUE_ MOTION or all targets given to the controller may be cancelled by CANCEL_MOTION. These services are provided in order to influence a manipulator’s motion by external signals.

In order to receive events from a controller, the RRS-Interface provides three services which cooperate with GET_NEXT_STEP. DEFINE_EVENT allows the programmer to define events with different modes dependent on combinations of time, traveled distance and position. GET_NEXT_STEP returns the number of events which occured during the last interpolation step. If any, detailed information about the events may afterwards be requested by the service GET_EVENT. With CANCEL_EVENT, one also has the possibility to cancel an event before it occurs.

Frames of reference

The RRS-interface supports a comprehensive controller internal kinematic model, which may be accessed and modified by the services GET_CELL_FRAME, MODIFY_CELL_ FRAME and SELECT_WORK_FRAMES.

Using cartesian position data, the movement of the robot can be defined by means of two frames. These are given by the ToolID and the ObjectID of the service SELECT_WORK_ FRAMES. The cartesian position data is the coordinate of the ToolID frame with respect to the ObjectID frame. As such, it defines the target to reach in the ObjectID frame by the ToolID frame.

The cartesian position data is defined with a homogeneous matrix which contains the position and the orientation values. The names for the elements are chosen according to /PAU81/

The vectors n,o and a describe the orientation frame and the vector p describes the position of a frame. The redundant fourth row is omitted.

nx	ox	ax	px	o : orientation vector
ny	oy	ay	px	a : approach vector
nz	oz	az	px	n = o x a
0	0	0	1	

Kinematic Models

Besides the two frames TOOL and OBJECT needed to describe the motion of a robot, some others are used to build or modify the geometrical description of a cell (figure 1.5).

Through the service MODIFY_CELL_FRAME, a programmer can modify the kinematic model of a robot's environment, which consists of a table containing the kinematic relations between the frames.

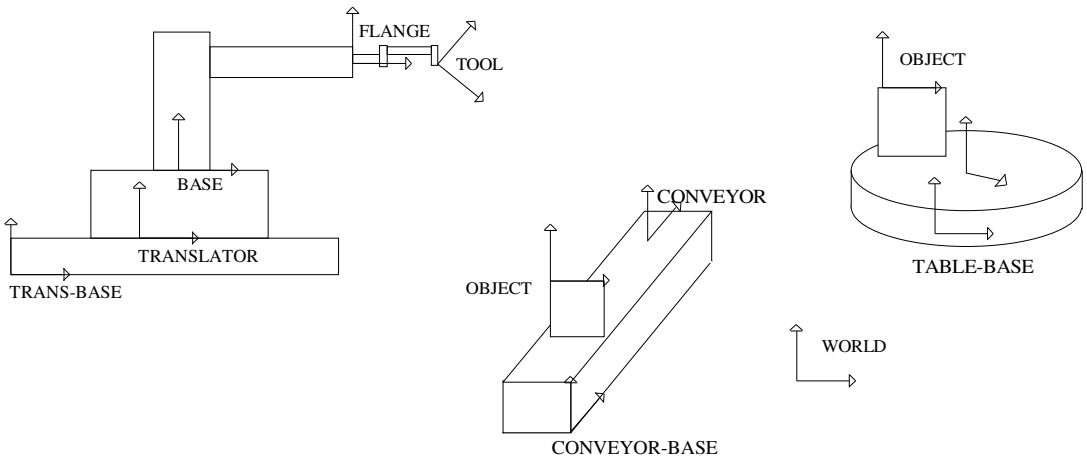


Figure 1.5 : Kinematic model /RRS95/

Frames are addressed by Frame IDs. Six IDs are reserved and have a unique meaning :

WORLD : This is the origin frame of the cell.
BASE : This frame represents the side of the robot arm which is attached to the WORLD
FLANGE : It represents the side of the robot arm which is attached to a tool or an object.
TOOL : This frame is reserved for tools, which can be attached to the FLANGE side of the robot or, indirectly, to the BASE side of it.
OBJECT : This frame is reserved for workpieces. They may be attached to the FLANGE or BASE side of the robot.
CONVEYOR : This frame is reserved for conveyor tracking. The robot controller doesn't control the path of the conveyor, but is merely synchronized with it.

Nr	FrameName	RelativeToName	FrameType	FrameData	Joint Nr.
1	TOOL	FLANGE	Constant	4x3	-
2	FLANGE	BASE	Robot	-	1 to 6
3	BASE	TRANSLATOR	Constant	4x3	-
4	TRANSLATOR	TRANS_BASE	Translate X	-	7
5	TRANS_BASE	WORLD	Constant	4x3	-
6	WORLD	-	-	-	-
7	OBJECT	TABLE	Constant	4x3	-
8	TABLE	TABLE_BASE	Rotate Z	-	8
9	TABLE_BASE	WORLD	Constant	4x3	-

Table 1.2 : Frames /RRS95/

Transformations

To let the CAR-Tools perform mathematical transformations in the same way as robot controllers do, RCS-Modules may provide the services GET_FORWARD_KINEMATICS, GET_INVERSE_KINEMATICS, MATRIX_TO_CONTROLLER_POSITION and CONTROLLER_ TO_POSITION_MATRIX.

The homogeneous matrix is a unique representation of a TCP location and its orientation. Though various algorithms can be used to define the relationship between a controller location and a transformation, these algorithms still exhibit the following problems :

- The definition of the algorithms may be ambiguous, such that one transformation may yield to multiple solutions or singularities.
- The controller may perform some additional hidden math that is not apperent or documented, including numerical round-off that is due to the precision differences between the CAR-Tool and the controller.
- The definition of the algorithms may change, depending on the controller version.

The services GET_INVERSE_KINEMATICS and GET_FORWARD_KINEMATICS provide transformations from cartesian positions to joint angles and vice versa. GET_FORWARD_ KINEMATIC computes the TCP-pose with respect to the defined reference coordinate system for a given joint position. These transformations will be performed with respect to the current tool and object frames selected by SELECT_WORK_FRAMES service.

Flyby and Point Accuracy

Flyby means that the controller can look ahead and take into account more than the current target when planning a path and calculating robot's motion. The biggest advantage of the flyby mode is that the corners can be rounded to maintain a constant speed.

The basic flyby service is the SELECT_FLYBY_MODE, used to turn the mode on and off. Furthermore, to fully support the flyby programming capabilities of robot controllers, SELECT_FLYBY_CRITERIA and SET_FLYBY_CRITERIA_PARAMETER services are defined to allow respectively the selection and setting of flyby parameters. To cancel any selected criteria, the service CANCEL_FLYBY_CRITERIA should be called.

Point accuracy defines a window which determines when the robot arrives at a target point. This accuracy can be programmed with SELECT_POINT_ACCURACY and SET_POINT_ ACCURACY services.

Motion Concept

As illustrated in the figure 1.6, SET_NEXT_TARGET and GET_NEXT_STEP are key services for motion.

Principally, SET_NEXT_TARGET is used by the CAR-Tool to supply the programmed target positions to the RCS-Module, one target at a time. Then, GET_NEXT_STEP is called repeatedly in order to receive information about robot's motion towards the target. Each time this service is called, a new interpolation step with the highest possible resolution is reported.

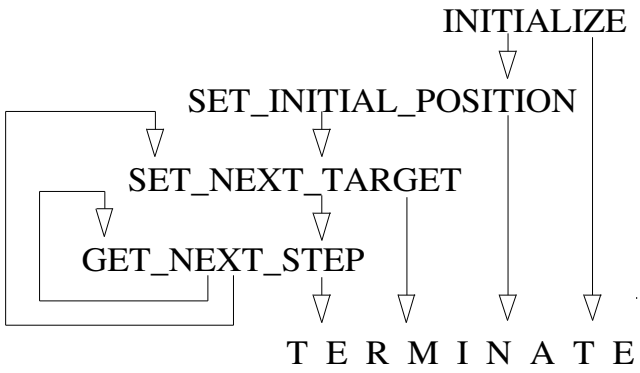


Figure 1.6 : The principle RRS-services

In this way, the targets are consumed by the RCS-Module, which may signal the need for new targets by means of the Status parameter. By using this output parameter, the RCS-Module has the possibility to look-ahead for an indefinite number of positions.

If the return status of GET_NEXT_STEP is 0, this means that the RCS-Module has calculated and is reporting a new position of the robot. In such a case, no new target is needed. Conversely, if the Status is 1, it should be understood that the RCS-Module needs more target data. In the latter case, an additional output parameter, namely *ElapsedTime*, is used to determine the validity of reported positions. If *ElapsedTime* has a value other than 0, a new position of the robot is reported, otherwise the RCS-Module is just asking for more target data.

If the *Status* of GET_NEXT_STEP is 2, it means that the RCS-Module has calculated and is reporting the final position of the robot. In this case, the robot is stopped either because the target is reached or as a result of a STOP_MOTION call.

To summarize the use of these parameters, an example where the path to be followed consists of four positions (P1 through P4, figure 1.7) may be helpful. Below is an imaginary robot program for this motion, accompanied with the corresponding RRS-service calls. After having reached the first target (P2), flyby mode will be turned on.

1	move to P1	SET_INITIAL_POSITION P1
2	move to P2	SET_NEXT_TARGET P2 loop : GET_NEXT_STEP (until Status=2)
3	flyby on	SELECT_FLYBY_MODE
4	move to P3	SET_NEXT_TARGET P3 loop : GET_NEXT_STEP (until Status=1)
5	move to P4	SET_NEXT_TARGET P4 loop : GET_NEXT_STEP (until Status=2)

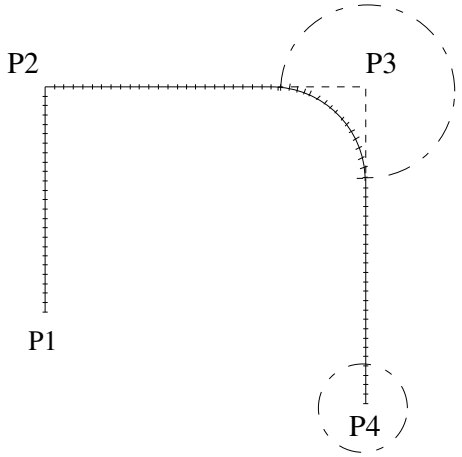


Figure 1.7 : Motion example

First, the initial pose of the robot and the target are set to P1 and P2 respectively. Then, the service GET_NEXT_STEP is called in a loop until it returns 2, indicating that the target has been reached. After turning the flyby mode on, the next target (P3) is supplied. Somewhere on the way to P3, the RCS-Module demands for the following target so that it can perform a corner rounding. At this point, P4 is supplied with SET_NEXT_TARGET again. Since the flyby mode is still on, the RCS-Module will report Status=1 before it reaches P4. This, however, will be ignored by the CAR-Tool which will keep calling GET_NEXT_STEP until Status=2.

Interrupting Motion

By using the service STOP_MOTION, it is possible to stop a motion which is currently in progress. As shown in the figure 1.8, when the RCS-Module receives this command, it generates a controlled, smooth deceleration until the simulated robot stops. The CAR-Tool has to call the GET_NEXT_STEP service further in order to get the interpolated position steps during this deceleration phase, until *zero speed* is reported from the RCS-Module. The STOP_MOTION command leaves the on-going motion in a resumable, pending state, in which the current target position is not cleared from the RCS-Module's memory and remains valid.

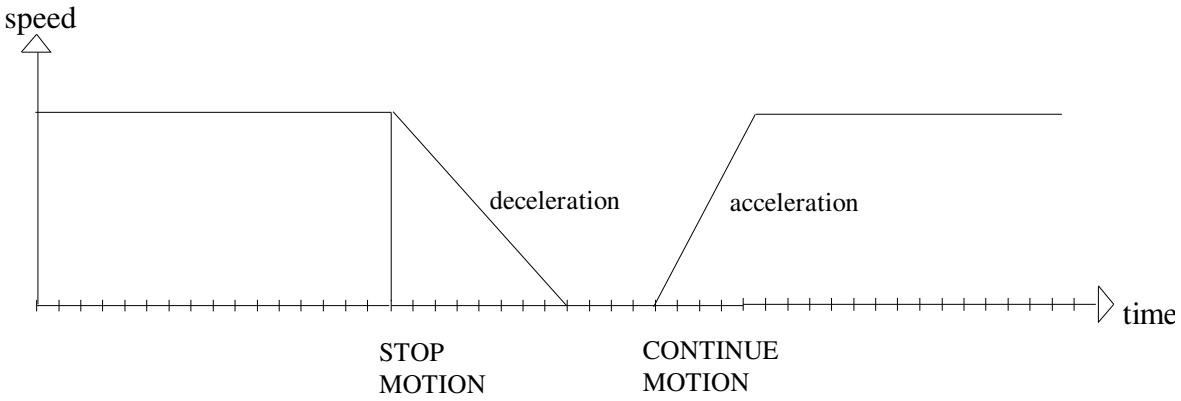


Figure 1.8 : Stopping and continuing motion

The CONTINUE_MOTION service restarts the last non-terminated motion that was stopped by the STOP_MOTION command. After the receipt of the CONTINUE_MOTION command, the RCS-Module generates a controlled acceleration for the simulated robot and considers the actual motion parameters such as speed and motion type for the continued motion. To get the interpolation steps belonging to the acceleration phase and to the rest of the path, GET_NEXT_STEP will have to be called again.

The CANCEL_MOTION service can be used after a motion is stopped by STOP_MOTION in those cases where the motion shall not be resumed. Since the current and all further target positions are cleared from the memory, the motion can no longer be resumed by CONTINUE_MOTION and is treated as completed.

1.3.2 RRS Calling Conventions and integration with CAR-Tools

The RRS-Interface requires the controller simulation packages from different controller vendors to be connectable to different CAR-Tool software packages. Both kinds of software packages may be written in several languages (C, Pascal, FORTRAN), are compiled with different compilers and have to run on different hardware platforms (e.g. HP, IBM, Silicon Graphics, SUN). Furthermore, the RCS-Modules need access to several operating system features (e.g. memory management, file access, multitasking, task communication) and finally, the RRS-Interface has to be extendible. In order to manage all these requirements and to reduce technical risks, rules for calling RCS-Modules and operating systems have been introduced /RRS95/.

The rules for calling RCS-Modules

- Each RCS-Module exports one function as its main entry point. The desired RRS-service is passed to this function as a parameter called OPCODE. This avoids linking problems if an RCS-Module doesn't support all services of the RRS-Interface. In case of an unsupported service, the RCS-Module must return an error status.
- All parameters of the services are passed within one Input-Data block and one Output-Data block. References to the Input and Output blocks are passed with the function call of the main entry point.
- The main entry is an ANSI-C function, which has the form

  void rcsx(void *in, void *out);

  x has to be substituted by an abbreviation of two characters denoting the RCS-supplier, to which two digits may be added for a version number. The parameters *void *in* and *void *out* pass the pointers to the Input and Output Blocks.
- The specification makes a distinction between Input/OutputBlocks and Input/OutputData :
  - The Input and OutputBlocks consist of a header, a parameter list and possibly a String-Data and unused space.
  - The Input and OutputData consist of the header, a parameter list and possibly String-Data.
  - The length of the Input and OutputData varies depending on the parameters of the service called.

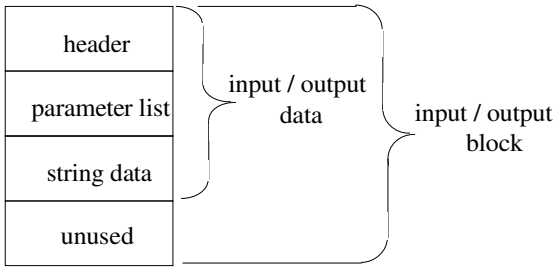


Figure 1.9 : The structure of the input and output blocks /RRS95/

The header of each InputBlock contains the following elements:

INPUT_DATA_LEN : The number of bytes actually used for input data (including the header).
OUTPUT_BLOCK_SIZE : The number of bytes that may maximally be used for OutputData.
OPCODE : The number of the desired service.

The header of each OutputBlock contains the following element :
OUTPUT_DATA_LEN :The number of bytes actually used for output data (including the header)

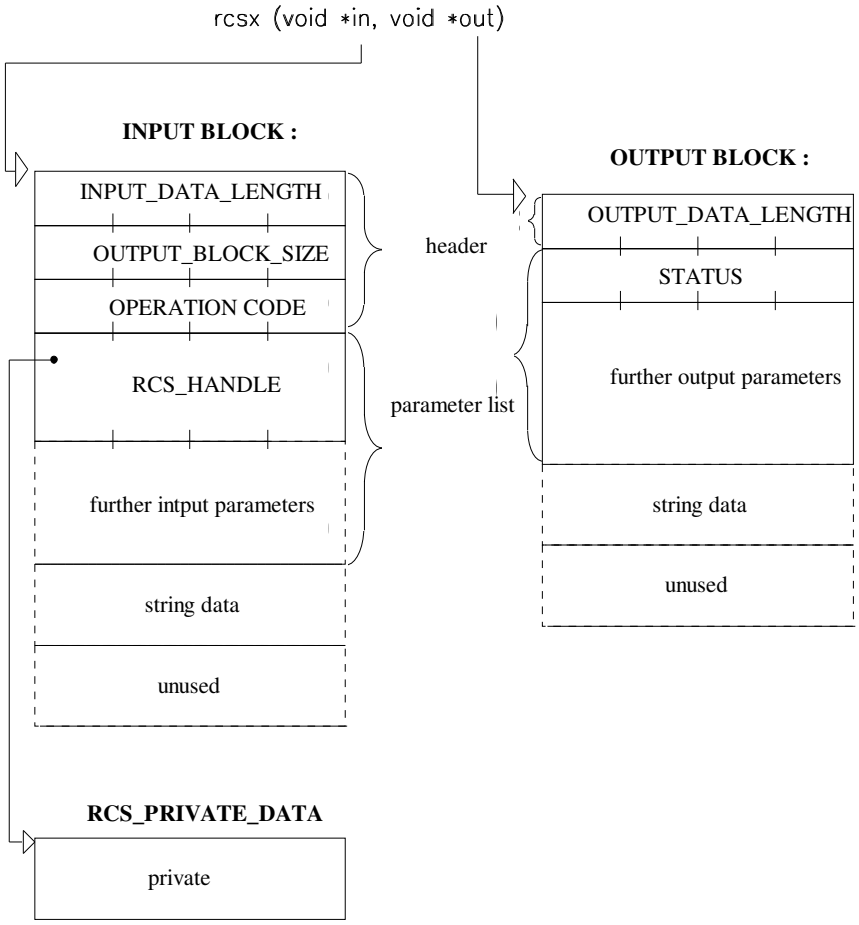


Figure 1.10 : Data structures of an RCS-Call /RRS95/

- All parameters of a service are transferred in the parameter list of a block. Each parameter has a fixed byte offset in the parameter list relative to the starting address of the block.

- A string is represented as a string-offset and a null terminated array of characters. The string-offset is placed in the parameter list and the null terminated array of characters is placed within the string data. The string-offset is the offset to the first character in the array relative to the start of Input or Output-Block.

- There are two special parameters which are used by all services :
  RCS-Handle : RCS-Modules may instantiate any number of robot simulations. An instance of a robot is created by an INITIALIZE service call. For further access to this instance, the INITIALIZE service returns to the CAR-Tool a field of eigth bytes called RCS-Handle. This is the first parameter in the input parameter list of all services except the INITIALIZE service.
  Status : Each RRS-service has this output parameter reporting the success/error status of the service.

The two module concept for integration

This concept assumes a communication line between the CAR-Tool and the RCS-Module. A typical solution may be realized with shared memory and semaphores.

Each time the CAR-Tool wants to call an RRS-service, it writes to the shared memory the contents of the InputBlock and raises a semaphore. The RCS-Module which waits on this semaphore reads the InputBlock, works, writes the OutputBlock to the shared memory and raises a semaphore for the CAR-Tool to read the OutputBlock.

Since this method does not require that the object codes are delivered to the user, each part can supply its module independently. Another advantage of this method is that problems of calling conventions from one language to another are avoided. The RCS-Module can be written in any language and the same is true for the CAR-Tool.

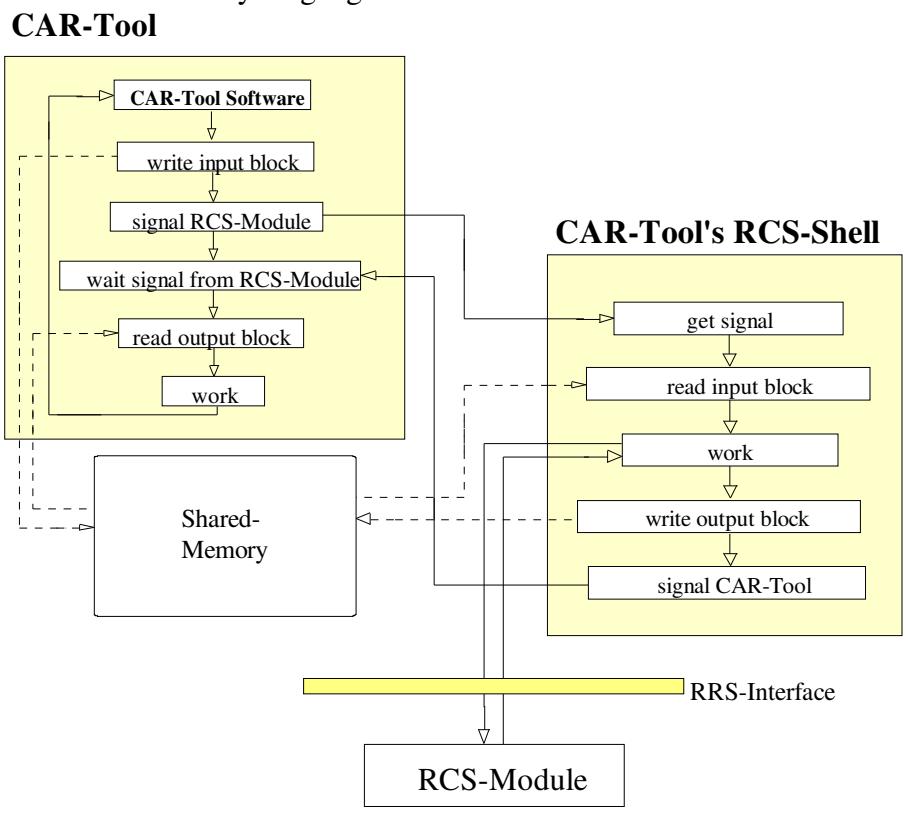


Figure 1.11 : Two-module concept for integration /RRS95/

2. Robotics at Volkswagen

2.1 VW as an industrial robot controller manufacturer

Volkswagen, besides being the first ranking automobile manufacturer in Europe, is also one of the most important robot manufacturers of the continent. Having been among the very first companies to plan and construct robotized production cells in the early 70's, it has presently got more than 6000 industrial robots, world-wide in use throughout the group.

In 1972, as the new VW product GOLF required an important variation in terms of models, single-purpose manufacturing equipments were to be taken out and replaced with more flexible production facilities. At this point of time, basing on earlier experiences with some USA-made industrial manipulators, the decision to develop and produce own robots was taken. During 20 years of independent work, more than twenty kinematical structures and three controller generations have been developed, by which top cost/performance ratios were achieved.

In 1993, for reasons of rationalization, VW made a contract with the company KUKA for the production and delivery of industrial robots consisting of Volkswagen controllers and KUKA mechanics. As a result, the costs for manipulator mechanics were reduced in average by 40%. Furthermore, the production of the controllers was taken on by the company SEF. In 1995, the design of a new robot controller (RCV) in cooperation with KUKA has been started. This new controller will be PC-based and run under Windows-95 and VxWorks.

VW	Wolfsburg	1092
	Hannover	577
	Braunschweig	246
	Kassel	119
	Emden	1006
	Salzgitter	12
	Mosel	184
	Brussels	194
	U. S. A.	74
	Mexico	6
	Brasil	17
	South Africa	17
	Pamplona	244
AUDI	Ingolstadt	1109
	Neckarsulm	677
SEAT	Martorell	402
SVW	China	2
Volkswagen	Group	5978
Sold		104
Total Production		**6082**

Table 2.1 : Industrial robots throughout the VW-Group

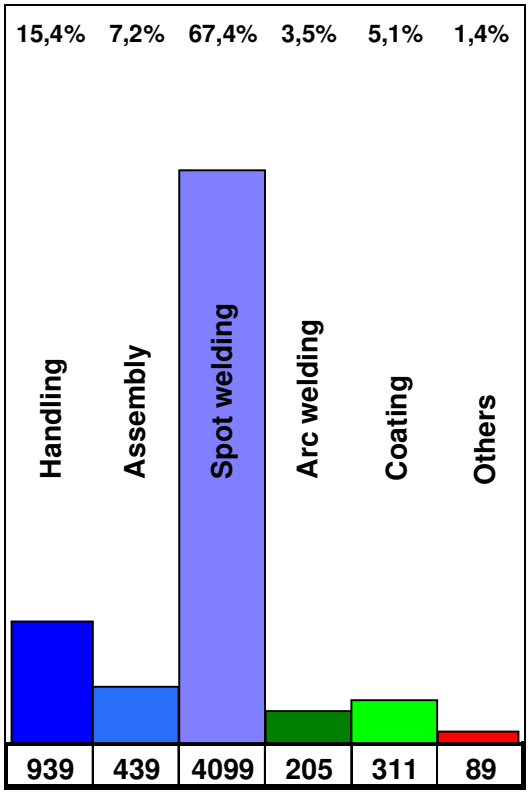


Figure 2.1:Application areas at VW

Manipulator type	Amount
K15, L15, R30, R100	825
G 8, G 10, G 15	460
G 60	832
G 80, G 100	627
G 120, G120A, G121	951
GP 100	129
P 30	20
P 50	260
P 60	22
P 80	8

Manipulator type	Amount
P 100	10
P 200	30
P Laser	8
S1, S2	98
VK 10	100
VK 30	46
VK 120	1270
VK 150	105
VK 360/125	281
Total	**6082**

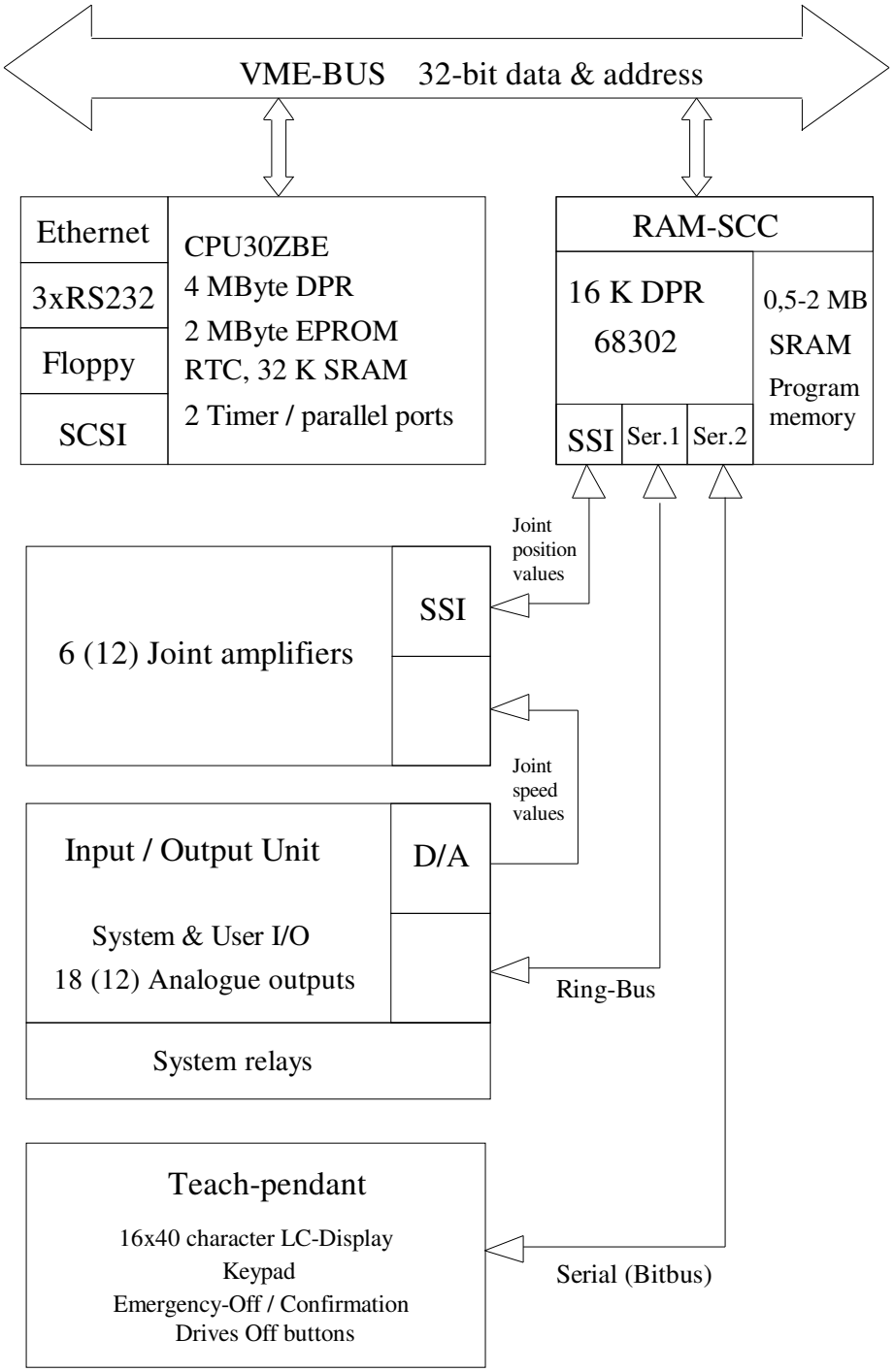
Figure 2.2 : Manipulator types used by the Volkswagen Group

Main features of the VRS1 controller

* 32-Bit Microprocessor (Motorola 68030 with co-processor 68882)
* Joint controller cycle-time : 5 ms (min), 15 ms (standard)
* Path module cycle-time : 10 ms (min), 15 ms (standard)
* Fast I/O Data-bus (1,5 MBaud)
* Absolute measuring systems, up to 24 serial bits
* Support of 12 joints
* Driver amplifiers (AC or DC) in 19" card format
* Distances up to 100m from the manipulator to the controller
* Data archiving via integrated floppy-disk
* Operation mode selection from the teach-pendant
* Integrated PLC functions
* Up to 128 Inputs, 128 Outputs, 127 Sequences, 127 sub-programs
* Multi-sensor guidance
* Programmable sensor interfaces
* Sensor informations taking effect in about 20 ms.
* Programmable analogue/serial/parallel interfaces
* Off-line programming (upload, download)
* Menu oriented teach-in programming
  - Sub-program technique
  - Conditional branching
  - Mirroring of programs
  - Interpolation types : Linear, Circular, Spline, Synchro-PTP, Twist
  - Weaving with all or only hand axes
  - Fly-by mode

2.2 Hardware architecture of a VW Robot Controller VRS1

The modules of a VRS1 robot controller (single CPU version) which are connected by means of a VME-Bus are illustrated in the figure 2.3.



7

Figure 2.3 : Computer architecture of the VRS1 controller

2.3 *Software structure of VRS1*

The Volkswagen Robot Controller Software VRS1 consists of many processes which run in parallel under the PDOS real-time operating system. Even today, its software structure still keeps the traces of its first implementation which ran on two CPU's, one for the path module and the other for the task submission and robot operation system.

With the exception of some sub-routines written in assembly, the software part of the controller is almost totally coded in the C programming language. The underlying real-time operating system is PDOS. VMEPROM consists of the real-time kernel of PDOS and is delivered with the controller. Other PDOS utilities are only used for development purposes.

1	Power-UP	Automatic loading of the operating system from EPROMS
2	Power-UP	Automatic loading of the path module from EPROMS
3	Power-UP	Automatic warm-up with the data recovered during the Power-DOWN
4	Power-DOWN	Automatic saving of the actual data (Sequence, point number and others)
5	Key-Diskette	The operating system controls access to protected data.
6	Floppy-Disk	Loading of the saved programs or constants from the floppy disk into the memory and SRAM disk.
7	Data saving	Saving of the programs or constants from hard (SRAM) disk to floppies.
8	Verify	Data comparison between the floppy and hard (SRAM) disks.
9	Verify	Data comparison between the RAM and hard (SRAM) disks.
10	Verify	Data comparison between the RAM and the floppy disk.

Figure 2.4 : VRS1 Controller Software /VRS1/

2.3.1 Features of the underlying operating system

VMEPROM is a PDOS based real-time monitor. The main features of its kernel are :

- Multitasking, multiuser scheduling
- System clock
- Memory allocation
- Task synchronization
- Task suspension
- Event processing
- Character I/O including buffering

The PDOS kernel is the multitasking, real time nucleus of the VMEPROM /VME89/. Tasks are the components comprising mostly a real time application. It is the main responsibility of the kernel to see that each task is provided with the support it requires in order to perform its designated function.

The most important responsibilities of the kernel are the allocation of memory and the scheduling of tasks, since each task must share the system processor with the others. The kernel saves a task's context when it is not executing and restores it again when it's scheduled. Among other responsibilities of the kernel, the maintenance of a 24 hour system clock, task suspension and rescheduling, event processing, character buffering and other support functions could be stated.

Task
A task is defined as a program entity which can execute independently of any other program if desired. From this point of view, it is the most basic unit of software within a real time kernel. A user task consists of an entry in the task queue, task list and a task control block with user program space.

From the time a task is coded by a programmer until its termination, it is in one of four task states. Tasks move among these states as they are created, begin execution, are interrupted, wait for events and finally complete their functions. These states are defined as follows :

Undefined : A task is in this state before it is loaded into the task list. It can be a block of code in a disk file or stored in memory.

Ready : When a task is loaded in memory and entered in the task queue and task list but not executing or suspended, it is said to be ready.

Running : A task is running when scheduled by the VMEPROM kernel from the task list.

Suspended : When a task is stopped pending an event external to the task, it is said to be suspended. A suspended task moves to the ready or running state when the event occurs.

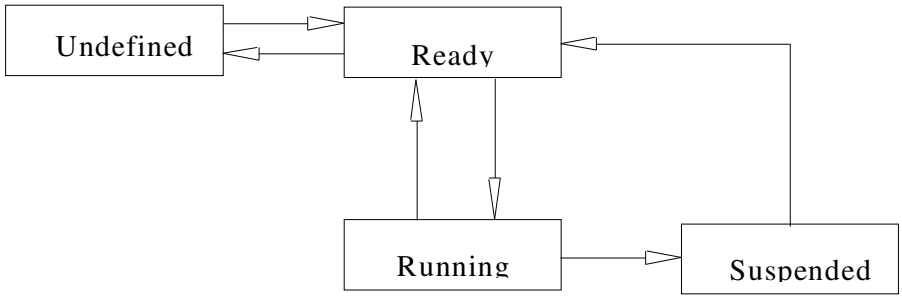


Figure 2.5 : Task states /VME89/

VMEPROM allows 64 independent tasks to reside in memory and share CPU cycles. Each task contains its own task control block and thus executes independently of any other task. Intertask communication and synchronization are integral parts of real time applications since many functions are too large or complex for any single task. For this purpose, the kernel uses common or shared data areas called mailboxes, along with a table of preassigned bit variables, called events, to synchronize tasks. A task can place a message in a mailbox and suspend itself on an event waiting for a reply. The destination task is signaled by the event, looks in the mailbox, responds through the mailbox and resets the event signaling the reply.

Events
Tasks communicate by exchanging data through mailboxes, whereas they synchronize with each other through events. Events are single bit flags that are global to all tasks. There are four types of event flags :

1 - 63	software
64 - 80	software resetting
81 - 127	system
128	local to task

Events 1 through 63 are software events. They are set and reset by tasks and not changed by the task scheduling. It is possible for a task to suspend itself pending a software event and then be rescheduled when the event is set. Depending on the application, one task must take the responsibility of resetting the event for the sequence to occur again.

Events 64 through 80 are like the normal software events except that the kernel automatically resets the event whenever a task suspended on that event is rescheduled. Thus, only one task is rescheduled when the event occurs. These events are set and reset by the Send Message Pointer (XSMP) and Get Message Pointer (XGMP) primitives.

Event flags
Event flags are global system memory bits, common to all tasks. They are used in connection with task suspension or other mailbox functions.

Message buffers
VMEPROM maintains 64 64-byte message buffers for intertask communication. A message consists of up to 64 bytes plus a destination task number. There is also the possibility to send more than one message to any task.

Message Pointers

VMEPROM supports shorter message pointer transfers between tasks with the Send Message Pointer (XSMP) and Get Message Pointer (XGMP) primitives. When a pointer is sent, the event *[destination message slot number + 64]* is set. Similarly, in case a message pointer is retrieved, the corresponding event is cleared.

Task suspension

Any task can be suspended pending one or two events. In such a state, a task will not receive any CPU cycles until one of the desired events occurs.

2.3.2 Software components of VRS1

Process name	Function
INTER1	Receipt of hardware interrupts generated by the co-processor, determination of the receiver-task and generation of a software interrupt
BMVERT	On-line interpolation and the handling of the on-line tasks
INIT	Motion planning
ERTV	Used near singularities, where the conventional inverse kinematic routines do not come up with an optimal behaviour
HTASK	Frequently needed computations, mostly transformations
SETINT	Conversion of the exception vectors of the MC68020-CPU 29 into those of the co-processor MC 68882
TRACE	Buffering of the *task* and *result* data structures for tracing
REGLER	Control of the joints
ROBO	Task-scheduling of the main CPU
HAND	Teach-in programming of the robot
HART	Critical real-time tasks in *Hand* operation mode
FYSY	Access to the floppy and hard disks
AUTO	*Automatic* and *Single-step* operation modes
SENSOR	Processesing of the sensor informations
SPS	Execution of PLC commands
MEMPRO	Control of some critical memory areas for security purposes
MEMVERW	Memory management on the main CPU
DISPPER	Output of the teach-pendant

Table 2.2 : Software components of VRS1 /HOC90/

2.3.3 The path module

The path module consists of the processes *INTER1*, *BMVERT*, *ERTV* and *INIT*. It is responsible for the trajectory generation, including the calculation of interpolation steps which the manipulator joints have to follow for a given motion. To this module are motion tasks to be submitted, which can be one of the following types :

- Start-task
- Continue-task
- On-line task

Basically, a motion task is a description of the start and target positions together with the motion type as well as a number of additional parameters. If the robot is not already in motion, a task of type *start* has to be submitted. Following the submission, the process responsible for the control of the joints will pick up the results from a special data structure until the path module reports that the motion has come to the end.

A task is submitted to the path module through a task structure, which includes the following informations :

Task code with motion type
Position informations
 TCP position
 Roll-Pitch-Yaw angles of orientation
 Joint values (12)
for
 Start point (used only with start-tasks)
 Target point
 Via point (used for circular& spline motion)
 Additional point (for spline motion)
Tool vector
Speeds
 at departure
 on the path
 at arrival
Acceleration at the beginning
Deceleration at arrival
Maximum jerk
Radius of the precision sphere
Weaving parameters
 Weaving form
 Amplitude
 Angle
 Plane
 Period
Sensor parameters

Figure 2.6 : Task data structure

The result data structure is described below :

Task code with motion type
Error
Flag variables
Last result
Fly-by mode active
TCP position
Roll-Pitch-Yaw angles of orientation
Joint values (12)
Joint and TCP velocities

Figure 2.7 : Result data structure

If the arrival speed at a target is other than zero, in order to have a smooth corner rounding at that point, the path module will need to know about the following target. A *continue*-task is used to submit motion tasks one step ahead, thereby making some informations (e.g. speed profile) belonging to the next motion already available for a proper fly-by behaviour.

An *on-line* task becomes particularly meaningful when the process in which the manipulator is involved has to take sensor informations into account and interrupts its motion. The action to be taken could simply be a velocity change or a jump to another programmed point or sub-program. *On-line* tasks are designed to cope with such situations and bring higher flexibility.

Below is an example for a typical program execution : At the beginning, the TCP is at the point P1. A *start* task is submitted to the path module with the coordinates of P2 as target. During the motion, before entering into the precision sphere covering the target, a *continue* task for circular motion is given. As the TCP moves towards P4, a second *continue* task is used to describe the next motion, which will be linear. At a point between P4 and P5, a sensor could indicate that the target object is near enough to halve the velocity for a smoother approach. For such a change, an *on-line* task is submitted.

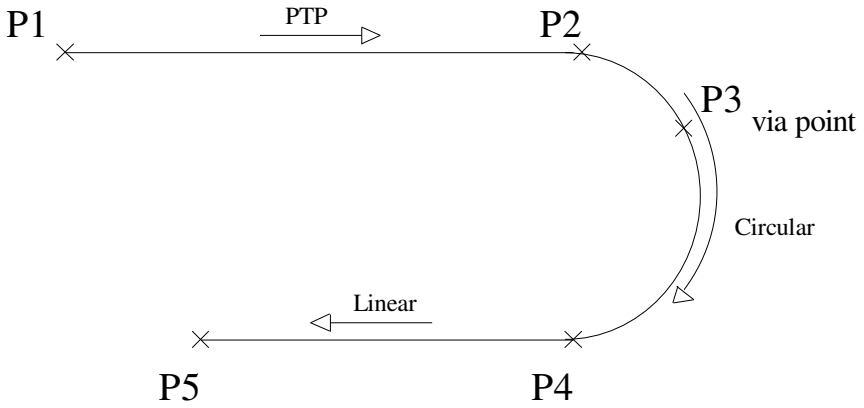


Figure 2.8 : Example of tasks

The figure below shows how motion tasks are submitted by the controller to the path module.

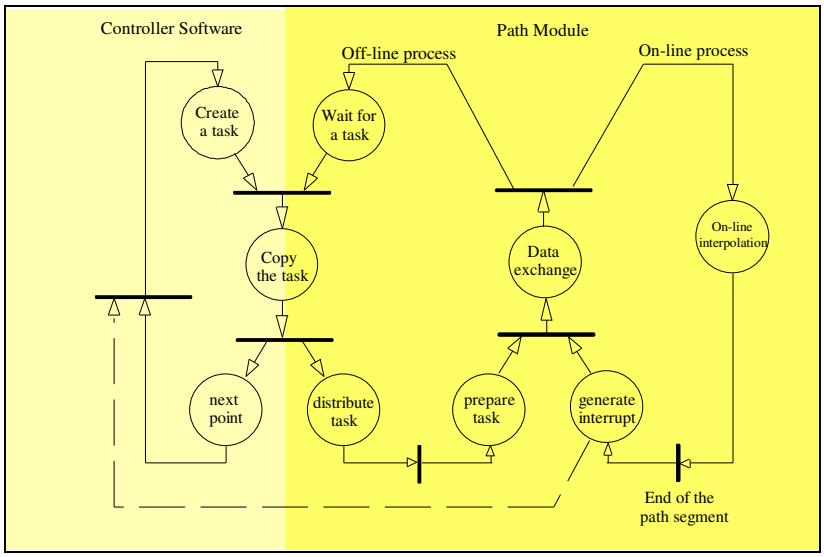


Figure 2.9 : Task administration in VRS1 /JAN91/

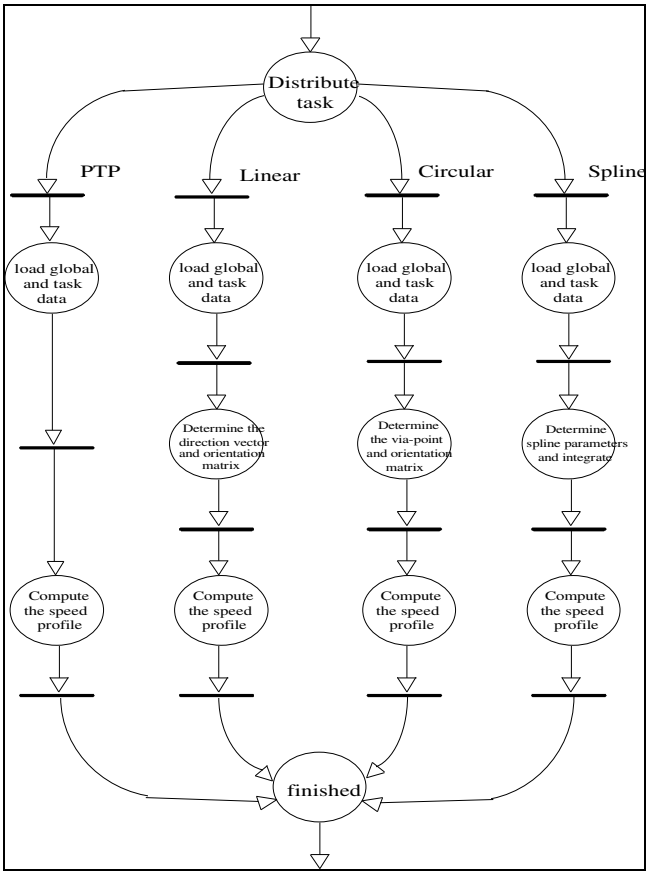


Figure 2.10 : Task preparation /JAN91/

2.4 Programming with VRS1

2.4.1 The teach-pendant

The basic element used for on-line programming is the teach-pendant. For the ease of programming, a teach-pendant should be available there, where the manipulator is planned to be used. This, however, would require that the controller-box stands near to the robot in the production line, where the space is usually quite limited and estimated at 5000 DM per squaremeter. For this reason, Volkswagen designed its manipulators in a way which allowed distances up to 100 meters between the controller and manipulator, letting the teach-pendant be plugged into the installation-box of the robot /BES95/.

By means of the VW teach-pendant (figure 2.11), programmers have access to all functions that the controller VRS1 offers. The basic elements of this device are :

- Display
- Function keys
- Joint motions/Coordinate systems keys
- Number block
- Drives On/Off
- Operating mode switches
- Approval/Emergency Off keys
- Joystick (optional)

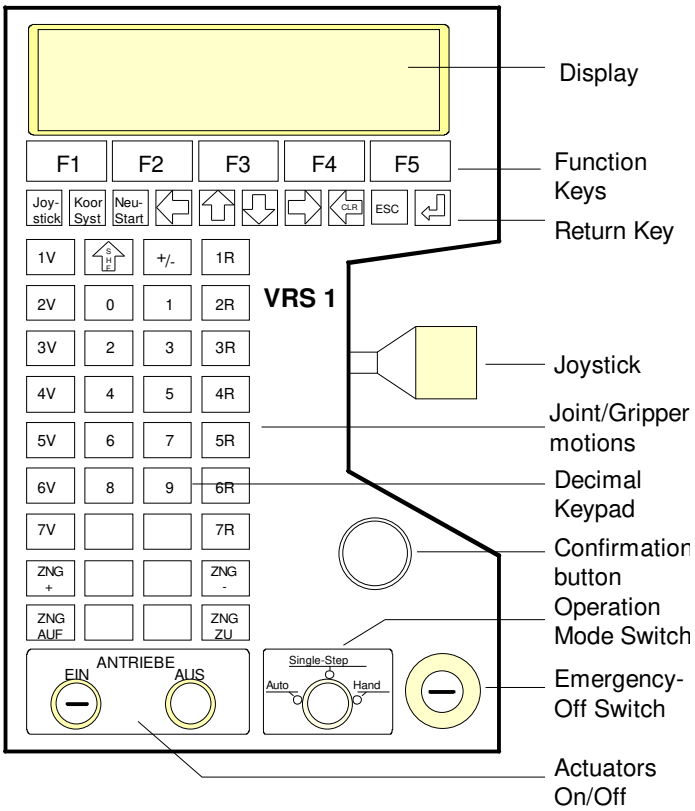


Figure 2.11 : The VRS1 Teach-pendant /VRS1/

Display
The communication between the programmer and the robot controller takes place through the display, which often consists of liquid crystal cells. All user inputs as well as system messages will be echoed on the display.

Function keys
There are three types of function keys. The first group is similar to the one on a personal computer, allowing the user to move the cursor to a desired location. To confirm the entry of a variety of parameters, the user is expected to press the enter key or switch the operation mode.

The second group of function keys guides the user through the menus, where he always has the possibility to jump from one parameter menu to another and to build up his program's flexibility.

By means of the third group, one has access to the tool's functionalities (e.g. grippers, paint guns or more complex assembly-devices) which are to be executed during the program.

Joint motions / Coordinate systems
One may use the joint motion keys to move each joint forward or backwards. The coordinate systems key allows to switch to a variety of motions as well as to select coordinate systems such as base, tool, flange, joint 1 and external.

The number block is used for numerical entries.

Drives On/Off
During the programming as well as in the automatic operation mode, this button may be used to turn off the amplifiers of the drives.

Operation mode switch
In the *Hand* operation mode, programs can be entered or edited. In this mode it is also possible to change system settings and run diagnostic programs.

In the *Single Step* mode, the robot executes a program under the control of two buttons. The release of one of these buttons will cause an immediate braking and stopping of the manipulator. As a security measure, the programmer must be keeping the *Confirmation* button pressed when he is present in robot's workcell to observe the program execution in real process speed.

In the *Automatic* operation mode, the manipulator will execute the selected program without pause. In this mode, nobody should be inside the protected zone which covers the workcell.

In addition to the robot manipulator, the whole equipment (transport band, elevators) can be switched off by pressing the *Emergency Off* button.

The joystick allows the motion of the TCP in cartesian coordinates inside workcell, thereby defining its speed by integrated potentiometers within the limits.

2.4.2 Teaching-in

The structure of a robot program can be divided into three groups :

- Geometry
- Additional informations
- Text

Each robot joint is equipped with sensors which deliver informations about the position, speed and motion direction of the joint. At the programming phase of the geometry, the manipulator is moved via joint-motion buttons or joystick of the teach-pendant to the desired position and joint encoder values are stored. These positions can be critical, in the sense that an action might have to be executed there (such as spot welding or gripping of an object), or they may only be auxiliary points inserted for a collision free path. During the programming, each stored pose is assigned a position and a predefined sequence number. These stored positions build up a robot program. Though the geometry of a program defines the desired robot positions, it does not specify how these positions are to be reached and what to do there.

As illustrated in the figure 2.12, each programmed pose includes the following informations :

- *Action :* In its widest sense, the action specifies the input/output values of the controller. These may be implemeted as gripper functions, on/off switching of actuators or output of analogue signals, as well as waiting for a condition before moving on. As with PLCs, these conditions may be combined in a number of ways as to achieve well defined dependencies.

- *Dynamics/Path geometry :* To this category belongs the interpolation type (PTP, linear, circular, spline, weave, twist), fly-by parameter (radius of the precision sphere), acceleration (also deceleration and jerk) and speed (beginning, path and arrival speeds) informations. Beginning from the first pose, all these parameters will be assigned their default values depending on the manipulator type, some of them will be automatically copied into the set of the following position.

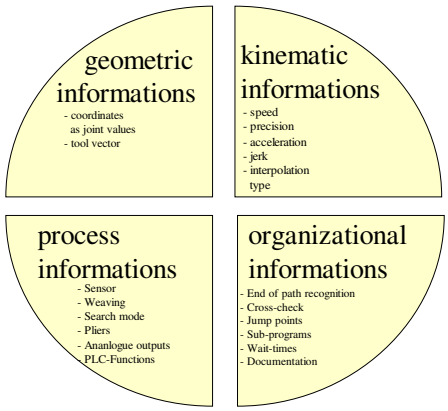


Figure 2.12 : Pose informations /VRS1/

- *Text :* This section offers an on-line documentation of programs for the use of maintenance personnel.

Testing of programs

After being taught, a program has to be tested before it is executed in the *automatic* operation mode. Tests are accomplished in the *single-step* mode, where the programmer has two choices: He can either execute the program with a maximum speed of 25 cm/min and let the controller ignore further security conditions, or use the *Confirmation* button to allow the manipulator to move in real process speed. This button is equipped with a spring which allows three different positions : Released, fully pressed or half-pressed. If the programmer does not keep the button in the half-pressed state, all drives will be switched off.. During the test, the biggest advantage of using real magnitudes in terms of speed will be the availability of realistic cycle times.

If a robot program does not meet the desired criteria at a specific point, the informations of this point can be directly edited.

The VRS1 controller supports also the shift, mirroring, rotation and copying of whole programs, which can be very beneficial in symmetrical production lines.

Fly-by capability

Usually, paths to be followed by robot manipulators consist of a set of curves in the space, at whose intersections different TCP speeds will be expected. Beside this, resulting sometimes from the motion type being used, the joint drives can be subject to unacceptably high acceleration rates in order to follow a predefined path.

The VW robot controller VRS1 allows a controlled fly-by at such points, whose crucial parameter is the radius *r* of a precision sphere. As shown in the figure 2.13, the controller will automatically change the radius of this sphere whenever the speed ratio is reduced. An important drawback of the fly-by mode is that the corners are only reached with a given degree of precision. If a point has to be reached as exactly as possible, the arrival speed to it must be programmed as zero.

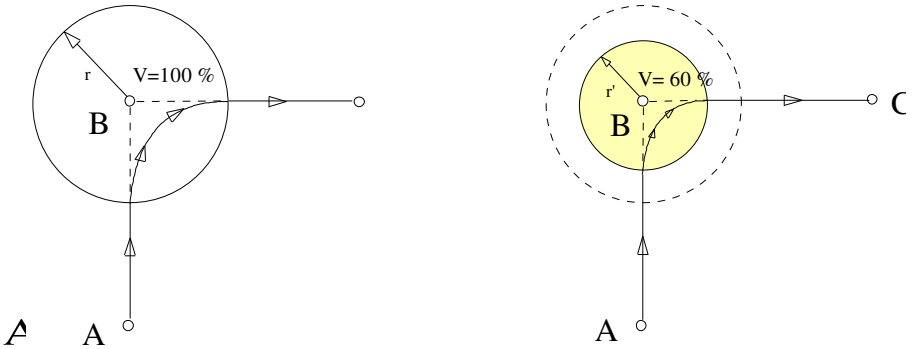


Figure 2.13 : Automatical change of the fly-by zone /VRS1/

3. RCS-Module for the Volkswagen Robot Controller VRS1

The responsibilities of the RCSVW-Module may be summarized as

- to communicate with the CAR-Tool in order to receive RRS service calls via input blocks and deliver output blocks for them.
- to interpret RRS commands and convert them into VRS1's internal format
- to allow simultaneous simulation of multiple robots by means of initializing as many controller softwares (path modules) as needed

For a better understanding of the data types and algorithms used by the RCSVW-Module, it would be helpful to have a closer look at each component of the path module to see their working and the way they communicate with each other.

3.1 The path module

3.1.1 Overview of algorithms

At the beginning of their execution, a common procedure followed by all components of the path module is to call an initialization function which will register their process identifiers into a global data structure called *system layout*. In this way, each process allocates a slot number for later use.

Secondly, each component initializes its local and some of the global variables. Finally, they all enter into their endless loops where they wait for specific software events signaling job-submissions for them. After working and producing output data, they all signal this situation by means of resetting the events they have waited for and suspend till the next job.

INTER1
The process INTER1 may be considered as the main entry point of the path module, because the RRS-Interface uses this process to submit any kind of motion tasks. Originally, its role is to handle hardware interrupts generated by the co-processor and give a further software interrupt to the related process.

BMVERT
This process is responsible for the calculation of interpolation steps during a motion. Before entering into its main loop, this program will first also check the existence of INTER1, INIT and ERTV by using message slots, and then wait for a reset task. No motion task will be processed unless these steps are properly taken.

In its endless loop, after being activated by the process INTER1, BMVERT will first wake up process INIT and wait until the latter computes a number of parameters indispensable for the on-line interpolation. Then, it will compute a motion step and generate a *result* event so that either the joint controller task of the real manipulator or the RCSVW-Module fetch the new joint values for motion. Before continuing, it will wait for the resetting of the same *result* event.

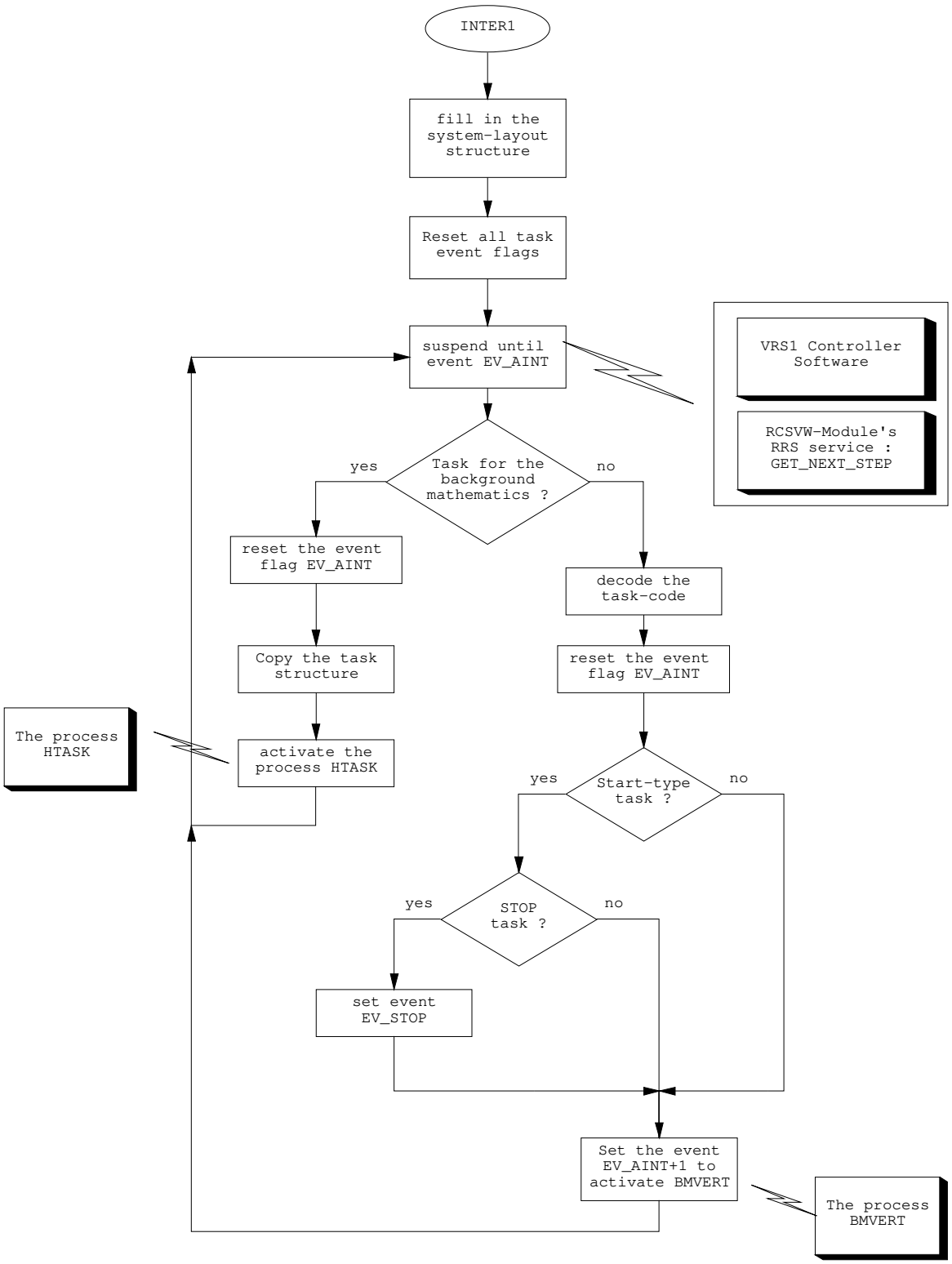


Figure 3.1 : Flow chart of the process INTER1

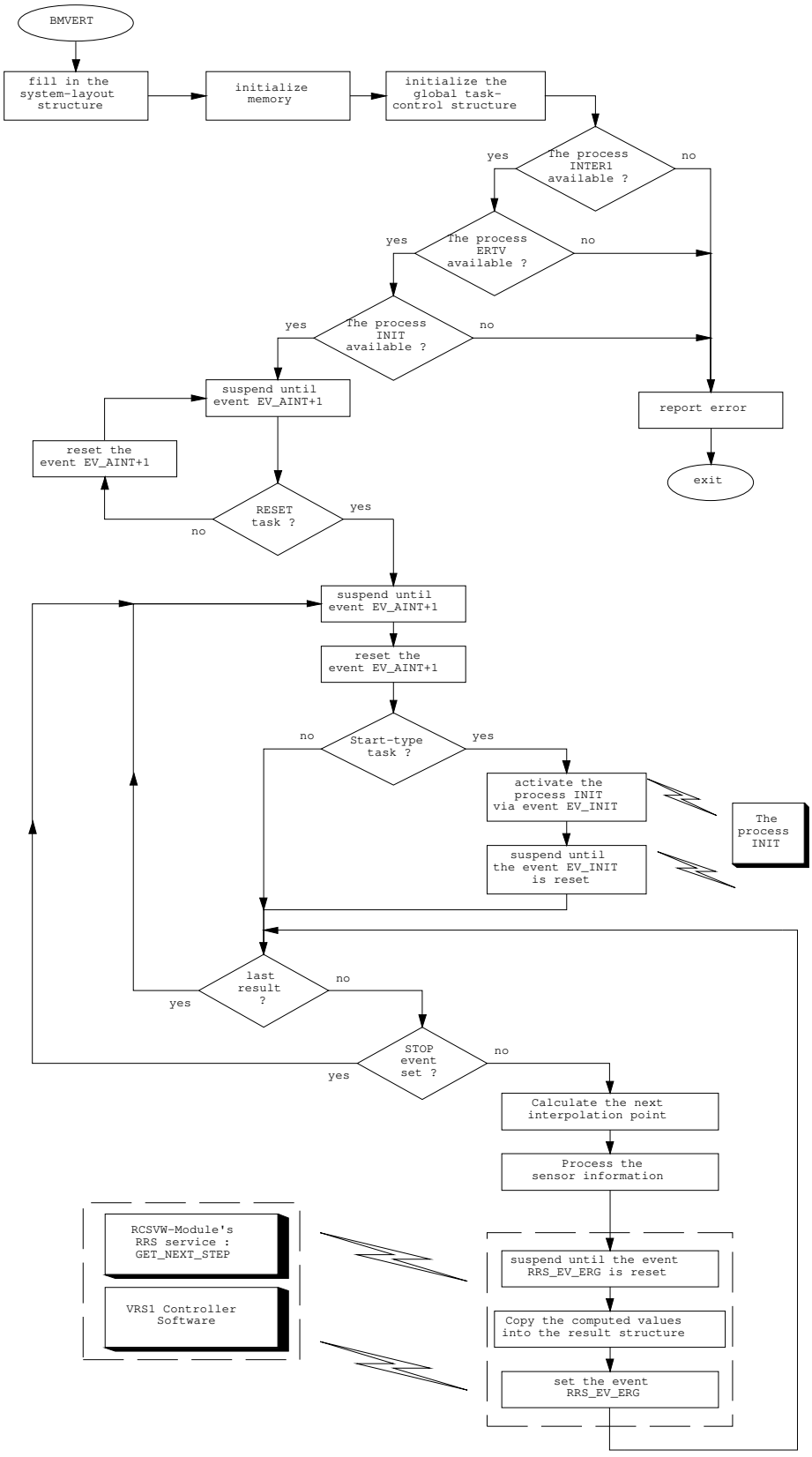


Figure 3.2 : Flow chart of the process BMVERT

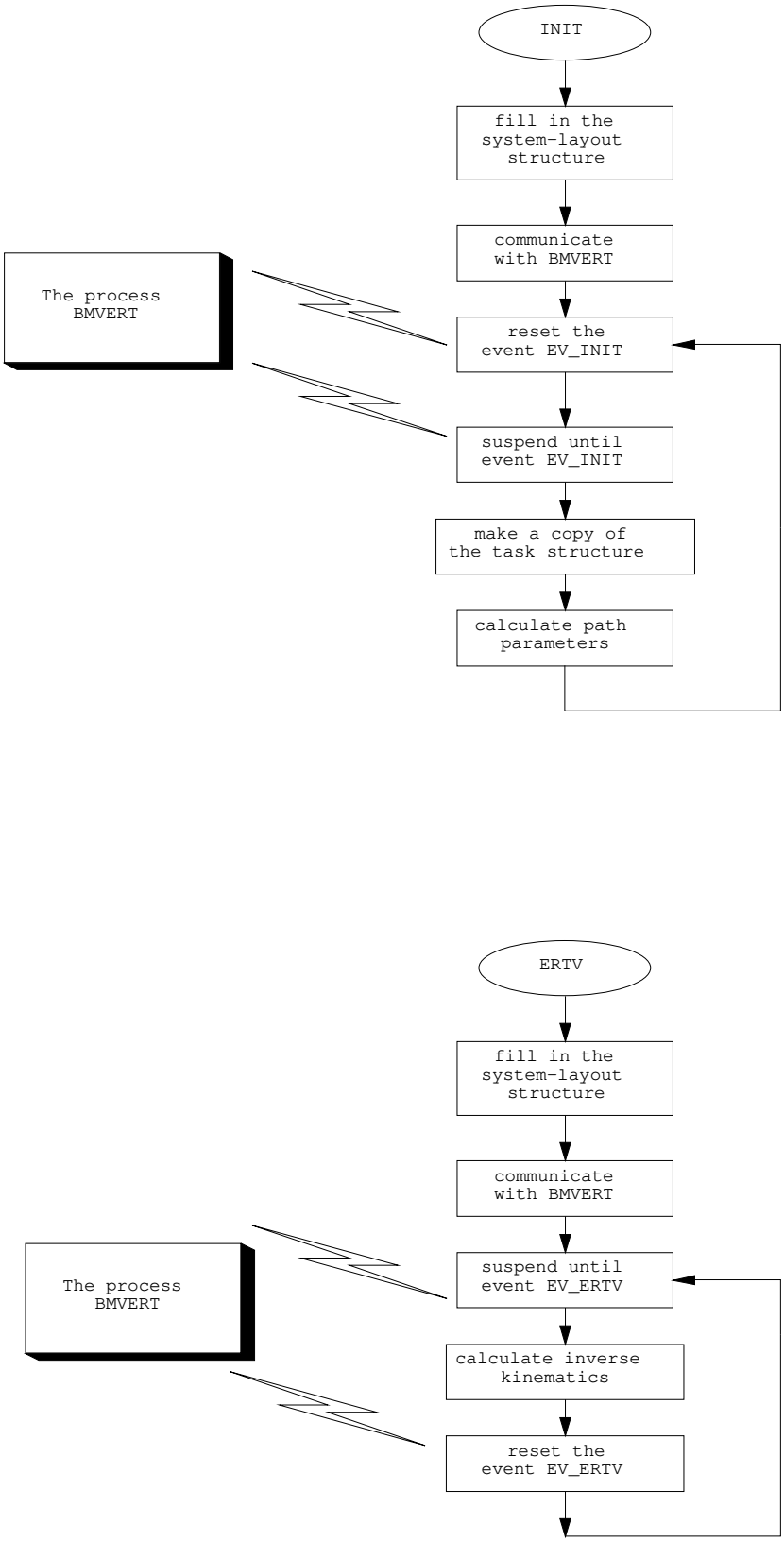


Figure 3.3 : Flow chart of the processes INIT and ERTV

During the design and development of the RCSVW-Module, particular effort has been spent to keep the original controller software as intact as possible. This, however, required the implementation of some interprocess communication tools of PDOS on a UNIX platform. For this reason, the first step has been to develop a set of communication services for the path module.

The most important advantage of such an approach is that the *original* path module becomes fully portable on UNIX compatible systems. Throughout its design, the RCSVW-Module has been extensively tested on both IBM RISC/6000 and Silicon Graphics Indigo machines, running with AIX and IRIX operating systems respectively.

Before describing the implementation of the library, it would be appropriate to give some basic information about the tools available on most UNIX compatible operating systems.

Interprocess Communication (IPC) tools under UNIX

There are several forms of interprocess communication, ranging from asynchronous signaling of events to synchronous transmission of messages between the processes. These mechanisms allow arbitrary processes to exchange data and synchronize execution /DAN91/ /BAC86/ .

Messages : The message type of IPC allows processes to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called *messages*. These can be sent or received by processes, which can also suspend their execution if they are unsuccessful at performing their operation. In other words, a process which is attempting to send a message can wait until the receiver is ready and vice versa.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. Each message consists of a message type, message size and text address as well as a pointer to the next message on queue.

Shared memory : Processes can communicate directly with each other by sharing parts of their virtual address space and then reading and writing the data stored in the shared memory. Processes may use system calls to create a new region of shared memory, to attach them to their virtual address space or to detach them.

Since the system calls for shared memory do not provide locks or access control among the processes, these must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another process is using.

Semaphores : The semaphore system calls allow processes to synchronize execution by doing a set of operations atomically on a set of semaphores. A semaphore is a positive integer, which can be incremented or decremented via semaphore operations. A process can test for a semaphore value to be greater than a certain value by attempting to

decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is smaller than it. While doing this, the process can have ist execution suspended until the semaphore value would permit the operation, or the semaphore facility is removed. The ability to suspend execution is called a *blocking semaphore operation*.

3.1.2 Development of an interprocess communication library under UNIX

In order to be able to let the original path module run on a UNIX platform, one needs to provide some necessary intertask communication procedures and related data structures. For the development of the RCSVW-Module, the only functions which needed to be implemented were the *xsev* (set event flag), *xtef* (test event flag), *xsui* (suspend until event), *xgmp* (get message pointer) and *xsmp* (send message pointer). Some other functions such as *xrts* (read task priority), *xstp* (set task priority), *asm* and *xpel* were written as dummy functions and have no functionality.

Though the system calls mentioned above do originally make use of system signals of VMEPROM, it has been observed that the standard message utilities on UNIX platforms were much more suitable for the development of a reliable set of communication services. For this reason, by using some additional data structures, the 'event signals' have been implemented as 'messages'. In this way, when a process pends an event, it does not actually make a *pause* system call, but gets blocked waiting for a specific type of message. Similarly, the setting of an event does not cause signals to be generated, but as many messages to be sent as needed.

To provide such a communication library, the RCSVW-Module needs two Interprocess Communication (IPC) facilities, namely a message queue and a semaphore set. Throughout the development of the RRS-Interface, a number of ways of allocating these IPC resources have been considered in many aspects and finally, the technique which will be described here has proved to be an acceptable solution.

At this point, it should be indicated that the types of the messages sent throughout these library calls play the most important role, whereas their contents are fixed and relatively short strings of rather symbolic values such as „*wake up!*“. The figure 3.4 illustrates the distribution of message types for different robot instances and event types.

Message_type = (RobotNumber * 1000) + Events_offset + 128 + event_type

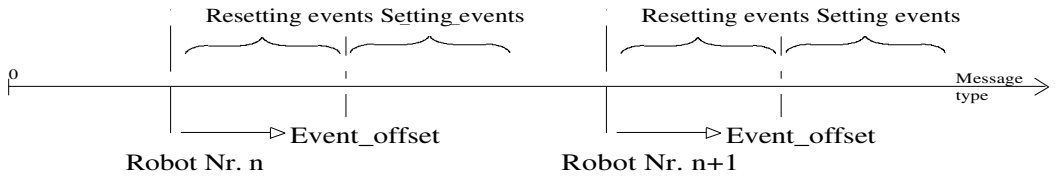


Figure 3.4 : Message types

Such a distribution allows the use of a single common message queue by many robot instances, each of them having their own event flags and message slots, expecting to communicate with the other members of their path module. As such, the message queue does not exhibit an important restriction in regard to the maximum number of robot instances which can be simultaneously instantiated.

An important point to be considered here would be the mutual exclusion of the library calls by many processes of a path module, which could access to global variables and change them simultaneously, leading to wrong actions. To prevent such inconsistencies, semaphores have been used to control access to event flags and message slots /BEN90/.

The RCSVW-Module has been designed to make use of only one semaphore set, whose variables are separetely dedicated to the use of single robot instances. Hence, the maximum number of robot instances which can be simulated in parallel is directly limited with the amount of semaphore counters available in a set. A typical value for the latter on UNIX systems is 25, being the lowest limit compared with the other system requirements of the module.

Event flags

Event flags under VMEPROM are global boolean variables, accessible by all processes running on the system. However, their most important function is the automatic waking procedure of the processes waiting for their (re)set.

An event flag has been implemented as a character variable together with an array of long integers. The *flag* variable is the state of the event flag, whereas the array *sleeping* is used to store the process identifiers of the processes pending them. Considering that the path module itself consists of four processes, the variable MAX_SLEEP_NUM has been defined as five. Indeed, the array *sleeping* could have been replaced with a simple counter variable, but has been kept to ease debugging efforts.

```
typedef struct {
        char flag;
        pid_t
sleeping[MAX_SLEEP_NUM];
        } event_flag_type;
```

If a process wishes to access any of the event flags by means of a library function, it first makes a semaphore operation to decrement the value of a binary semaphore counter (P operation). After finishing its work, it increments this value again (V operation). In this way, it can be ensured that processes will not enter into critical code areas at the same time.

xsev (set event flag)

The setting of an event consists of two steps. First, the *flag* variable is set to its new value. Then, the *sleeping* array is searched for nonzero entries to find out about the number of processes waiting for messages of this event type. If such an entry is found, a *msgsnd* system call is used to send messages of a specific type in order to wake up these processes.

xsui (suspend until event)

In case a process wants to pend an event, it first checks the state of that event flag. In case the flag has already got the desired value, the function will simply return, letting the process continue its execution. Otherwise, the function will look for a free *sleeping* array cell to register its process identifier and then make a *msgrcv* system call to receive a message, whose type is a function of the robot instance being involved, the event number as well as the event type (set or reset). In this way, the process can be blocked until such a message is put into the message queue.

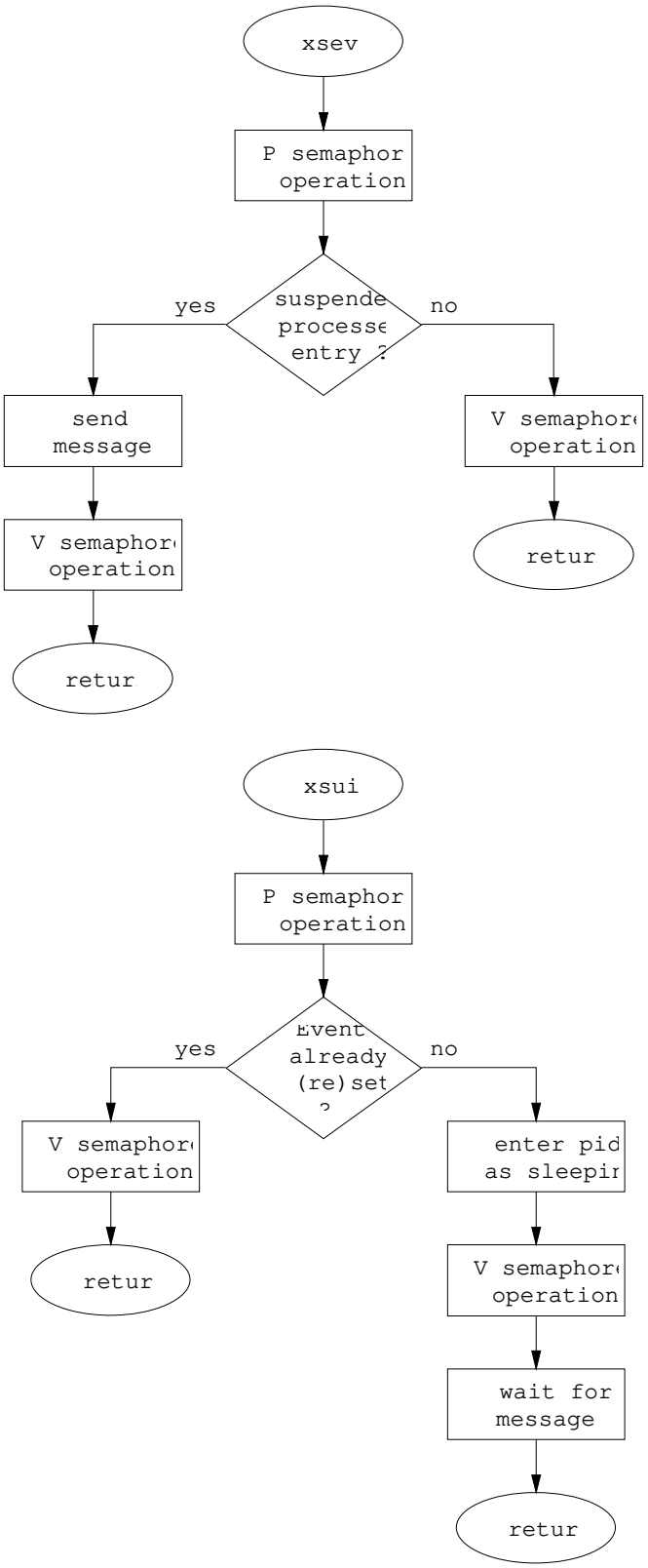


Figure 3.5 : Flow charts of the communication functions xsev and xsui

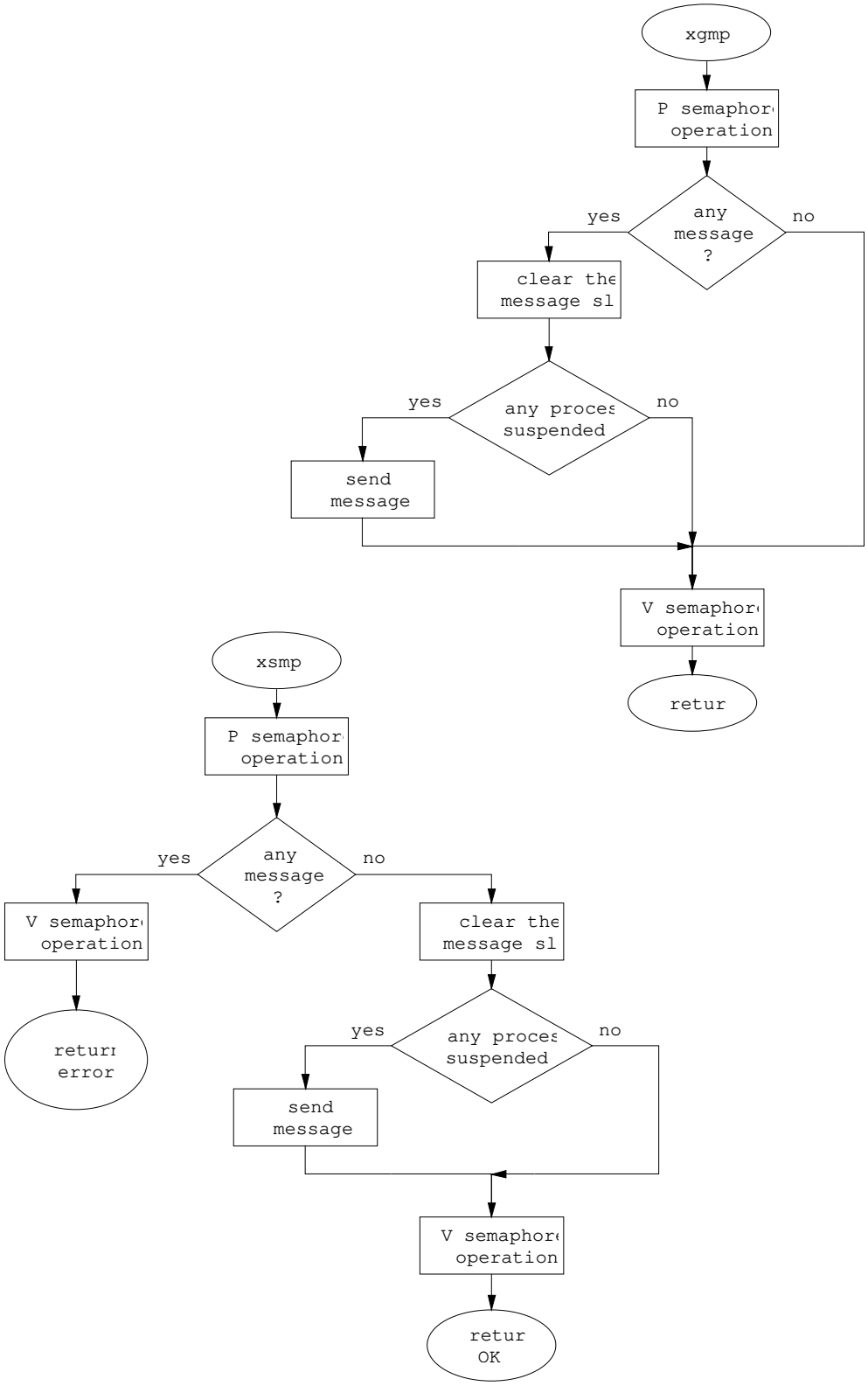


Figure 3.6 : Flow charts of the communication functions xgmp and xsmp

Message slots

Similar to event flags, message slots are also global variables. They have been implemented using the following data structure :

```
typedef struct {
        pid_t taskid;
        void *message;
        } message_slot_type;
```

xsmp (send message pointer)

This function will assign the process identifier of the calling process to the *taskid* variable and store the message pointer argument into the *message* variable. After this, an event of type (64+slot number) will be generated to wake up pending processes using the *xsev* function.

xgmp (get message pointer)

The function *xgmp* reads the *message* variable of a specific message slot and returns its contents. If the message slot is empty, an error value will be returned.

3.1.3 Memory requirements of the path module

Originally, the path module makes use of a *system layout* data structure which determines the beginning addresses of other data structures. With this technique, there is only one absolute memory address that the controller software (or the RCSVW-Module) has to determine, namely that of the *system layout*. The individual processes making up the controller software do all acquire this address by means of *xgmp* (get message pointer) calls, making them fully independent of specific memory partitionings which can be found on a variety of hardware platforms.

As it can be seen from the data structure below, the controller software accomodates the necessary data structures and modularity to support up to four simultaneous path control. However, this feature has only been used for test purposes so far and has never been implemented for industrial use.

Array of 30 integers	The message slot number that process will use
Array of 30 integers	The process identifier
Array of 30 integers	The process number (internal to the module)
Array of 30 integers	Number of the path module to which the process belongs
Arrays of 5 integers and pointers	Trace pointers and path module numbers for trace-buffers
Array of 4 pointers	Pointers to the global data structure of each path module

System-layout data structure

Although this method brings a high degree of flexibility, processes on a UNIX platform have to make some additional system calls to attach the shared memory segment to their virtual address space. Under UNIX, since the shared memory segments are identified by a system-wide valid integer value, the RCSVW-Module passes this identifier to the components of the path module as an argument of the *execlp* call. The next step of these child processes is therefore to make a *shmat* (attach shared memory) call for gaining access to this segment.

The shared memory segment of each robot instance is illustrated in the figure 3.7.

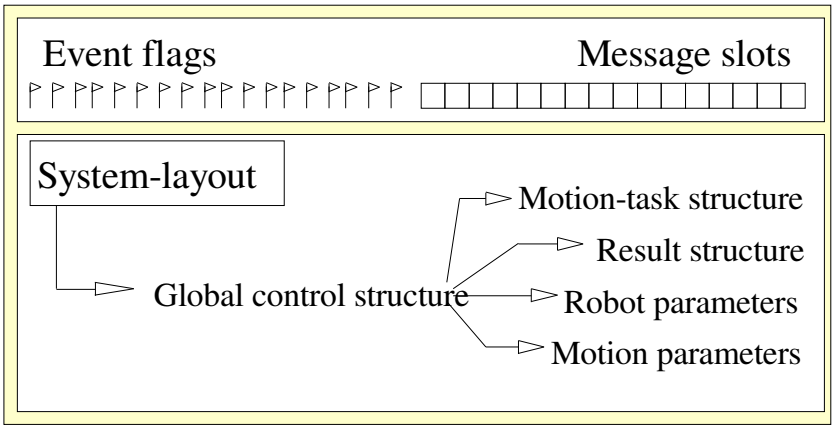


Figure 3.7 : Shared memory segment of a robot instance

3.2 RCSVW internals

3.2.1 Data structures

As described in the Realistic Robot Simulation Interface Specifications, an RCS-Module must be able to initiate as many robot instances as required by the CAR-Tool and support simultaneous simulation of all of them. For this purpose, the RCSVW-Module keeps track of its robot instances by using a data structure of type *robot instance*, in which the following informations are stored :

- The stamp of the robot
- Process IDs of the processes making up the path module
- The identifier of the shared memory segment
- The beginning address of the event slots in the shared memory
- The beginning address of the *system layout* data structure
- Control flags for motion
- The kinematic model of the robot
- Current OBJECT, BASE and TOOL Frames
- Matrices for OBJECT->BASE transformations and vice versa
- Control data for debugging

The robot stamp

The stamp of a robot instance consists of three character strings /RRS95/:

Manipulator type : This string is provided as a parameter by the CAR-Tool during the INITIALIZE service and determines the robot data file to be opened by the module. The file name results from the concatenation of the RobotPathName and the ManipulatorType according to the following rule :

filename = RobotPathName + "r" + ManipulatorType + ".org"

Originally, this robot data is available as an ASCII file on the key-diskette of the controller. The advantage of this nomenclature is that the present set of available robot data becomes totally accessible to the CAR-Tool without having to define any additional data types for the RRS-Interface.

example :	RobotPathName	: /local/robcad/dat/
	ManipulatorType	: vk010
	filename	: /local/robcad/dat/rvk010.org

Controller version : This string has a fixed value, which is "VRS1".

Software version : In the same way as it is done in the original path module, this parameter is read from a file named "version" under the ModulePathName directory. In the RCSVW-Module, the date information is also appended to the version string. (e.g. "VERSION001.9 (date : 20.05.94)") In case such a file can not be found, default values will be assigned to the related variables.

Process IDs of the components

The process IDs of the child-processes may be extremely useful, in that they provide the RCSVW-Module with a means of checking whether the components of a path module still exist or not. In case one of these processes or the operating system generates a kind of termination signal, the RCSVW-Module must be able to realize this situation, handle it properly and report it to the CAR-Tool at the next call. For this reason, each RRS-service call is preceeded by a test of all sub-components of the path module being involved. This is accomplished by generating *kill* system calls with null signals, which will perform normal error checking but will not send any signal /STE92/. If any of these system calls fail, other processes belonging to this path module will be terminated and the shared memory segment of that robot instance will be removed from the system.

Control flags for motion

To keep track of some events which are internal to the controller software and to ensure a proper submission of motion tasks, a control bit-flag variable has been introduced. The 10 least significant bits of this flag variable and their use are described below.

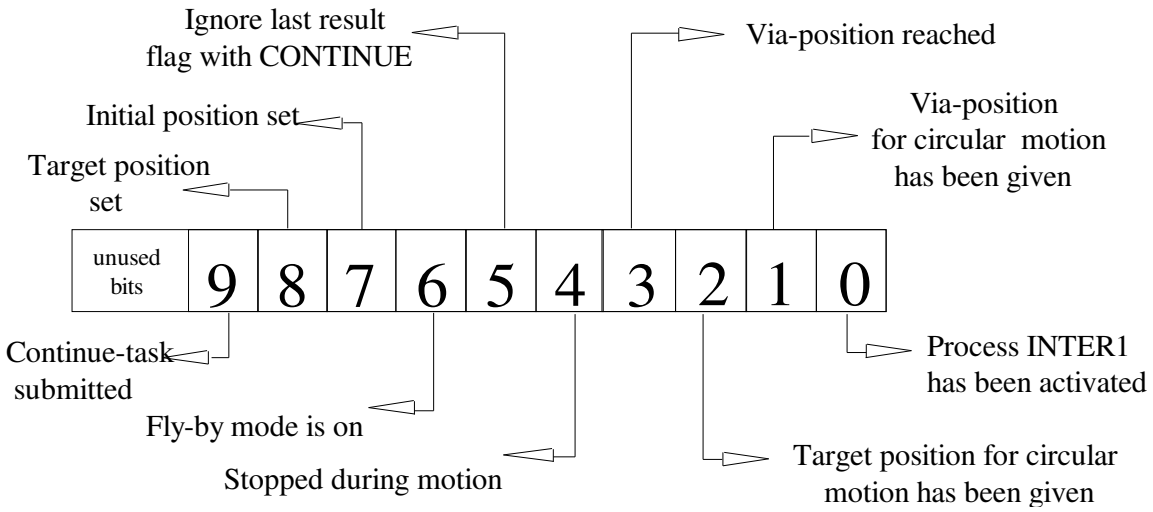


Figure 3.8 : Motion control bit-flags of a robot instance

- The process INTER1 has been activated.

This bit-flag shows that a motion task has been submitted to the path module by means of activating the process INTER1. As illustrated in the flow chart of the RRS-service GET_NEXT_STEP, the normal procedure following such a submission would be to wait for the event RRS_EV_RESULT, and then to fetch the result from the result structure. After this, depending on the value of the *last result* variable, either this flag or the event RRS_EV_RESULT would be reset to pick up another interpolation point.

- Via-target position (for circular type motion) has been given.

If the motion type is circular, two target positions are needed to define the path unambiguously. This flag will make it certain that no motion tasks will be submitted until these two target poses have been set via SET_NEXT_TARGET service.

Target position (for circular type motion) has been given.
Since the order in which the two targets are given is definitive for the selection of the right arc to follow, this flag is used together with the via-target position flag.

- Via-position (during circular motion) has been reached.

During circular motion, the path module will report that the result is the last one even if only the via-target position is reached. However, this result status has to be ignored until the target position is attained. By using this flag, the RCS-Module can find out whether the last result report is due to via- or target position.

- Robot stopped during motion

This flag enables the module to realize that a motion task has been stopped by the STOP_MOTION service. The service CONTINUE_MOTION can be successfuly used when this flag is set.

- Ignore the 'last result=2' when continuing

As a result of the piping of result structures in the path module, it is possible to receive additional *last result* reports after a CONTINUE_MOTION task, which indeed belongs to the last terminated motion. This flag will allow to ignore these results.

- The fly-by mode is on

Though the fly-by mode becomes automatically active whenever a precision sphere is defined or the end-speed has a value other than zero, this flag has been introduced for future extensions.

- The target position is set.

This flag will be set whenever a target position is set by the SET_NEXT_TARGET service.

- A continue-task has been submitted.

In case the conditions for fly-by mode are fulfilled, the path module will need to look one target ahead. In such a case, a *continue* type task is used to submit the task belonging to the following motion. Using this flag, the GET_NEXT_STEP service will be able to report that it needs more target data by returning status 1.

- Initial position has been set.

Before any motion, the initial position of the robot has to be given with the service SET_INITIAL_POSITION.

The kinematic model of the robot

Since the original controller software did not make use of any of the frame representations defined by the RRS-Interface, a simple kinematic model has been introduced for the support of related services. The most important advantage of this model is that the CAR-Tool can get the interpolation steps as joint values as well as cartesian position (and orientation) data.

The kinematic model consists of the BASE, OBJECT and TOOL frames, all defined with respect to the WORLD frame. These frames are illustrated in the figure 3.9

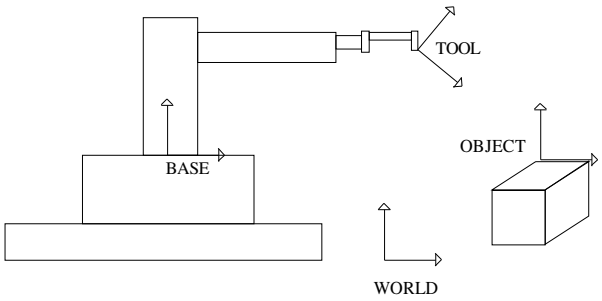


Figure 3.9 : Kinematic model

Current OBJECT, BASE and TOOL Frames

These frames can be read or modified by the CAR_Tool through RRS-services such as GET_CELL_FRAME and MODIFY_CELL_FRAME.

Matrices for OBJECT->BASE transformation and vice versa

By definition, the cartesian position data are the coordinates of the TOOL frame with respect to the OBJECT frame, in other words, it defines the target to reach in the OBJECT frame by the TOOL frame. However, since the controller software reports the TCP position merely in the BASE frame, one needs to convert the results into the desired frame coordinates.

This is done by a transformation of the TCP position and orientation into the TOOL frame. As it can be seen from the kinematic model table given above, the BASE, OBJECT and TOOL frames are all defined with respect to the WORLD, which necessitates two transformations. For computational efficiency, a third frame has been introduced to combine these two tranformations and is updated with the frames.

Control data for debugging

Debug-control data is an array of *debug_service_type*, containing the operation codes of the services which can be debugged and the number of times each service has been called with debug option.

For all supported RRS-services, additional debug functions have been developed to enable an easy debugging whenever necessary. Hexadecimal as well as extended debug format can be selectively used with all of them. Examples of log files in both formats can be found in the appendix.

As suggested in the RRS-Interface Specifications, the log-files are always opened in append mode for each RRS service call and closed after usage. This method enables debugging personnel to browse these files at any time between RRS calls.

3.2.2 Initialization of the RCSVW-Module

The RCSVW-Module has been designed basing on the *two module concept* introduced in the RRS-Interface Specifications. As a result of this approach, the interface itself consists of an executable shell which has to be spawned by the CAR-Tool.

The executable shell is called by the CAR-Tool with three arguments : Two identifiers belonging to the shared memory segment and semaphore set, and an offset value. Since the shared memory segment is used for input and output blocks, it must first be attached to the RCSVW-Module. The offset value which is passed as the third argument indicates where the output block within this memory segment begins.

The RCSVW-Module itself makes use of semaphores and message queues. Therefore, its first step is to set a signal handler which can catch a number of termination signals and use this occasion to kill its child-processes, remove their shared memory segments and release the semaphore as well as the message queue from the system before exiting. After this, internal data structures for robot instances are initialized.

Finally, the RCSVW-Module enters into an endless loop where the semaphore synchronizes the access to the shared memory segment.

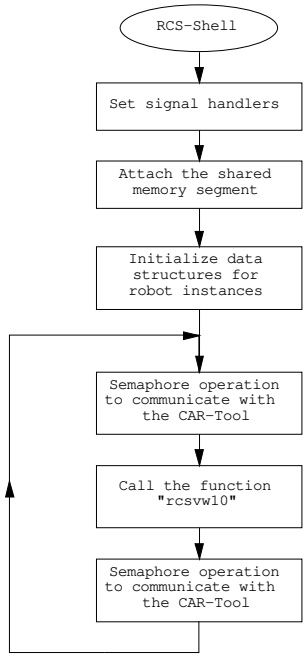


Figure 3.10 : Flow chart of the RCS-Shell

The requirements of the RCSVW-Module with regard to the system resources may be summarized by the following table :

	RCSVW-Module	**per robot instance**
Number of semaphore sets	1	none
Number of semaphores in a set	25	none
Number of message queues	1	none
Number of shared memory segments	-	1
Size of the shared memory segment	-	66 Kb
Number of processes	1	4
Maximum number of robot instances	25	

Table 3.1 : Requirements of the RCSVW-Module

The main entry point function : *rcsvw10(void *in, void *out)*

The function *void rcsvw10(void *in, void *out)* is the main entry point of the RCSVW-Module. As defined in the RRS-Interface Specifications, the two pointers *in* and *out* are pointing to the input and output blocks for the RRS service being called.

Once called, the function *rcsvw10* checks first the validity of the operation code indicated in the input block data. Secondly, the *RCS-Handle* variable (with INITIALIZE, the *RobotNumber* variable) is examined to see whether the robot instance being handled is a valid one. After having found out for which robot instance the service was called, some global variables are updated in case of necessity and the existence of the sub-processes of that specific path module are tested. On success, the operation code is finally used as an index to an internal table of RRS-functions and the desired service is called with the same *in* and *out* pointers.

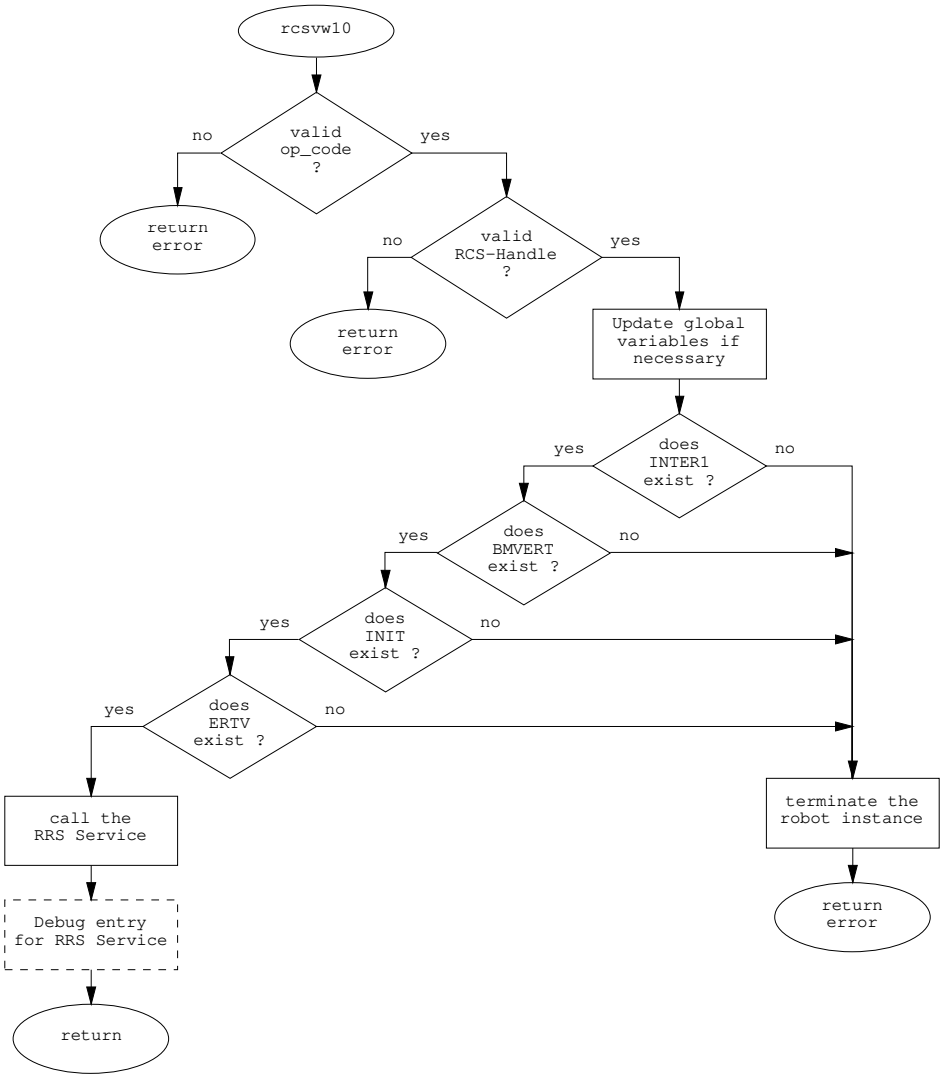


Figure 3.11 : Flow chart of the function 'rcsvw10'

In case the debugging mode is on, the operation code will be used again as an index to a table of debug functions. Since each RRS service receives and delivers specific input and output parameter lists, additional debug functions have been developed to support log entries with extended format. An example of such a file may be found in the appendix.

3.2.3 Algorithms of basic RRS-Services

INITIALIZE

This service initiates a robot instance by means of spawning a path module with all four processes and providing them with a shared memory segment for their use. If any child process does not respond within a given period of time after being spawned or in case the *reset* task does not succeed, the service will return an error status.

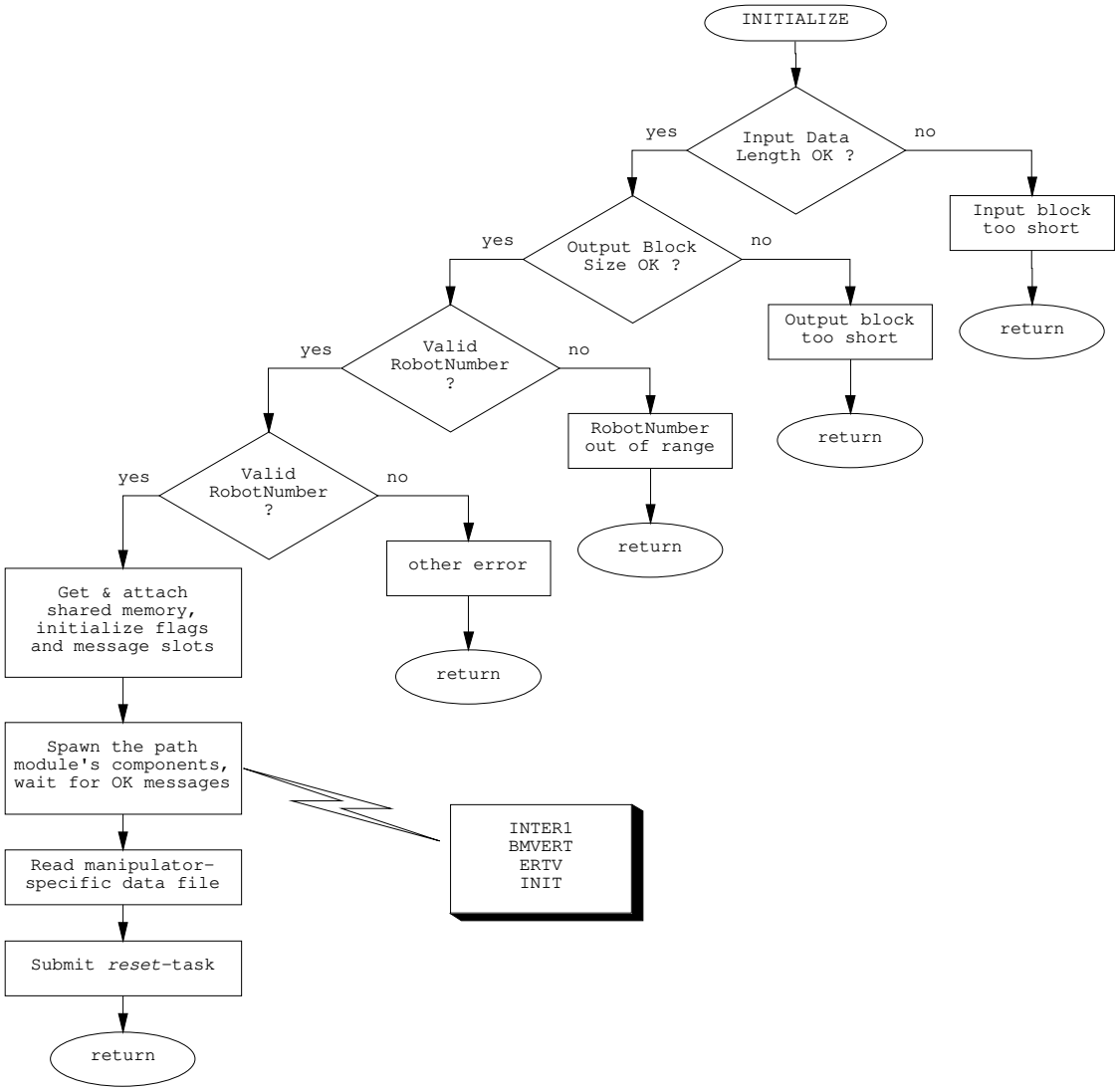


Figure 3.12 : Flow chart of the RRS-Service INITIALIZE

SET INITIAL POSITION

This function merely copies the joint values to the *task* structure, without submitting any kind of motion task to the path module. The conversion of the joint angles from radian into encoder values is accompanied with a check of joint software limits.

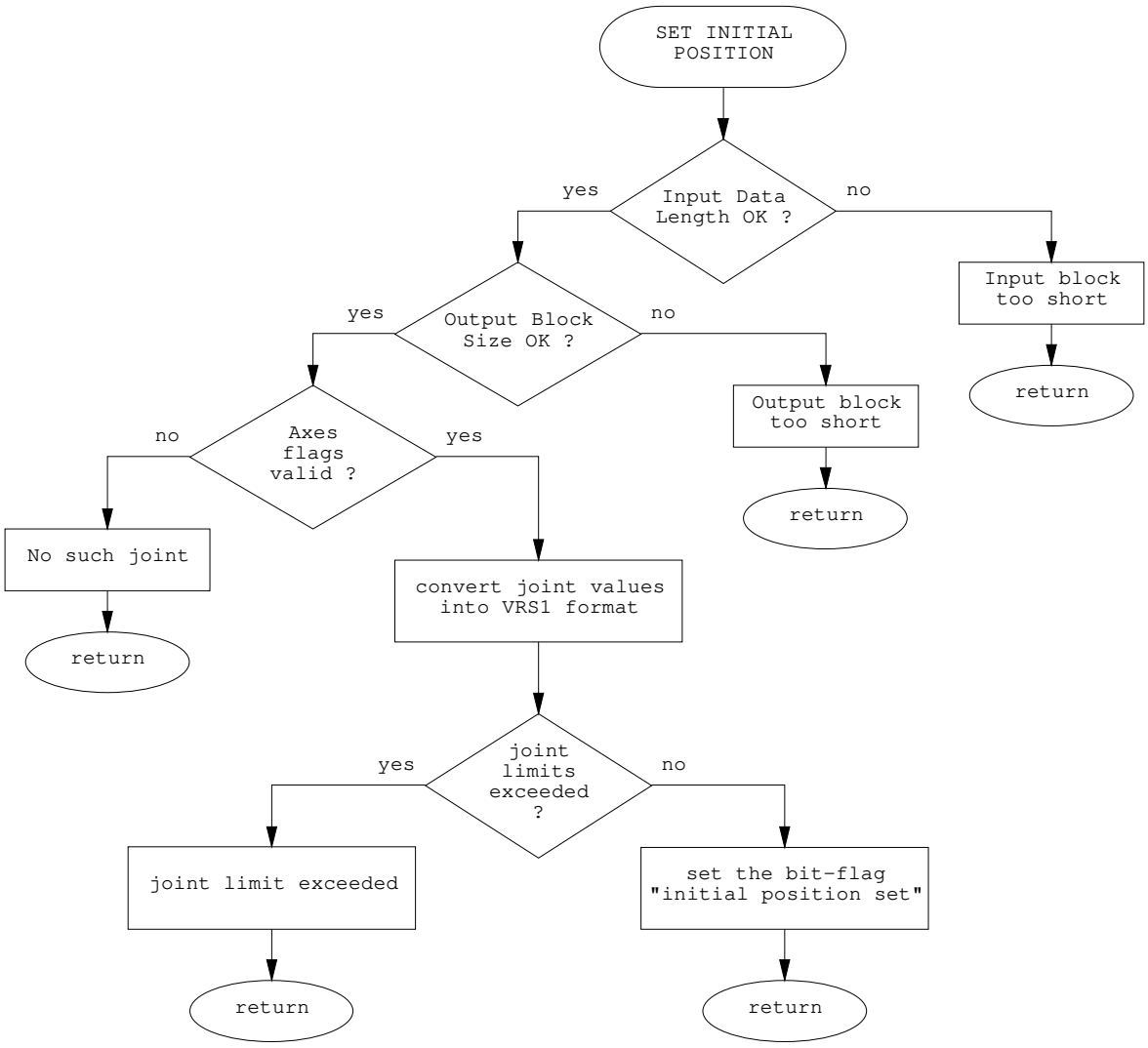


Figure 3.13 : Flow chart of the RRS-Service SET_INITIAL_POSITION

SET NEXT TARGET

This service is used to copy the joint coordinates of the target position into the task data structure. For circular motions, the first available target will be considered as via-position and the following ones will be set as main targets.

In case there is an on-going motion and the fly-by mode conditions are satisfied, this service will also incorporate the submission of *continue* motion tasks.

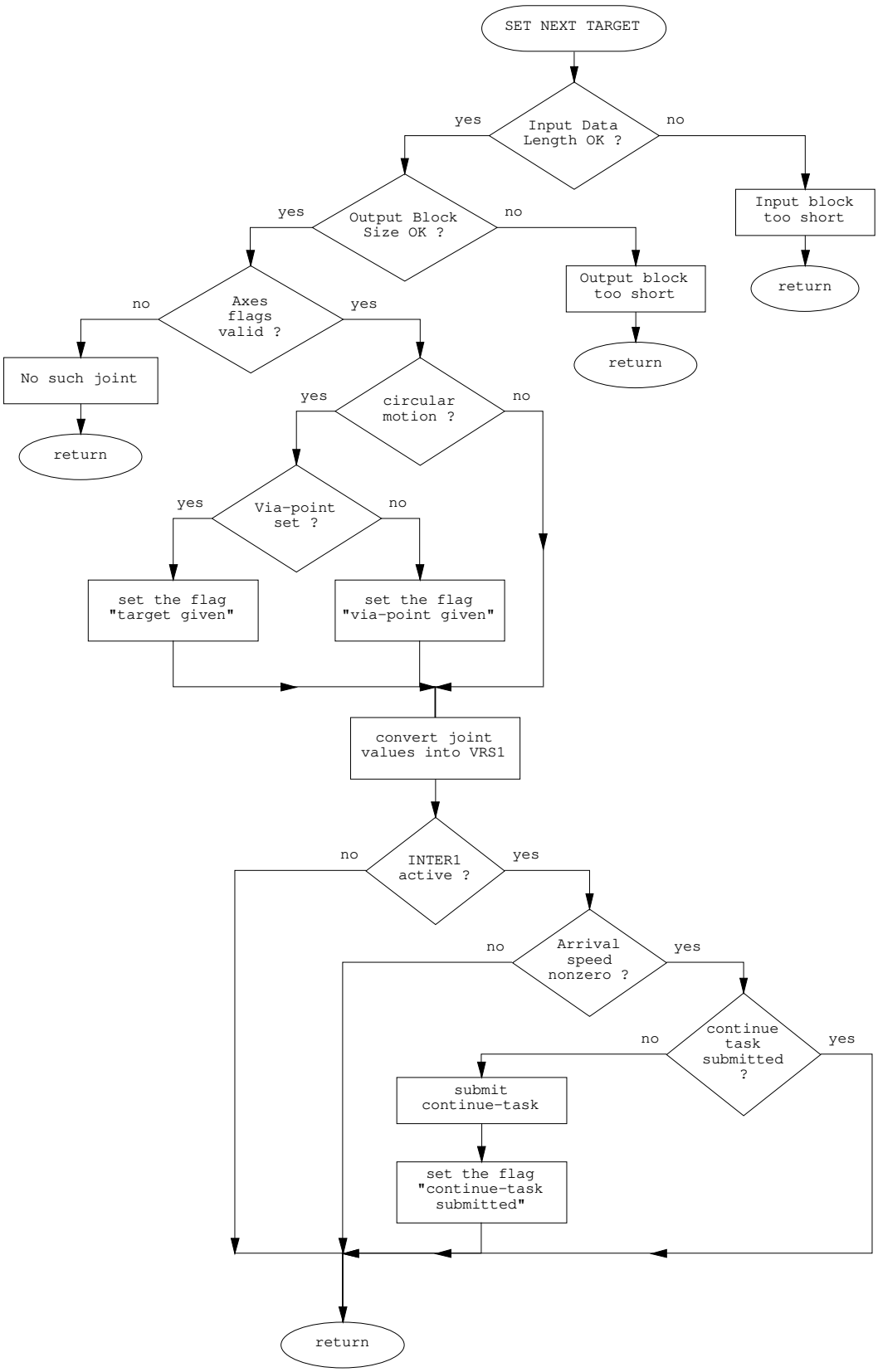


Figure 3.14 : Flow chart of the RRS-Service SET_NEXT_TARGET

GET NEXT STEP

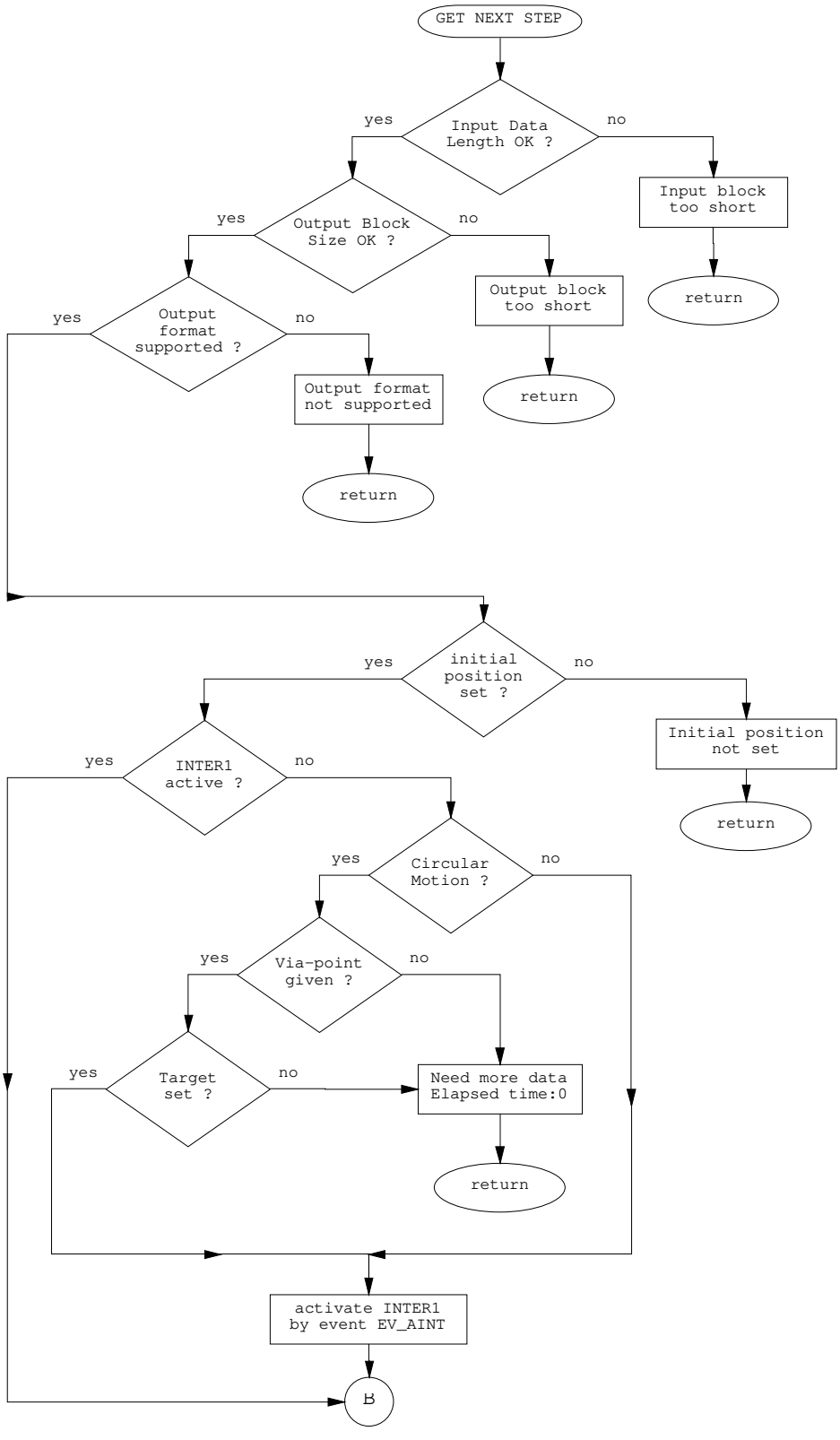


Figure 3.15 : Flow chart of the RRS-Service GET_NEXT_STEP (Part I)

The service GET NEXT STEP is responsible for both submitting motion tasks of type *start* and fetching the results delivered by the BMVERT process. The task submission is realized with the event EV_AINT, whereas the event RRS_EV_ERG is used to signal the availability of the *result* data structure for new interpolation steps.

This service uses the bit-flag „INTER1 activated“ to find out whether there is already an on-going motion or not. In case the manipulator is in standstill, a new motion task may be submitted. Otherwise, it will have to wait until the path module sets the event RRS_EV_ERG to signal the availability of a new set of joint values.

For circular motion, three additional bit-flags are used to interpret the results of the path module accurately and differentiate between a number of situations which could occur.

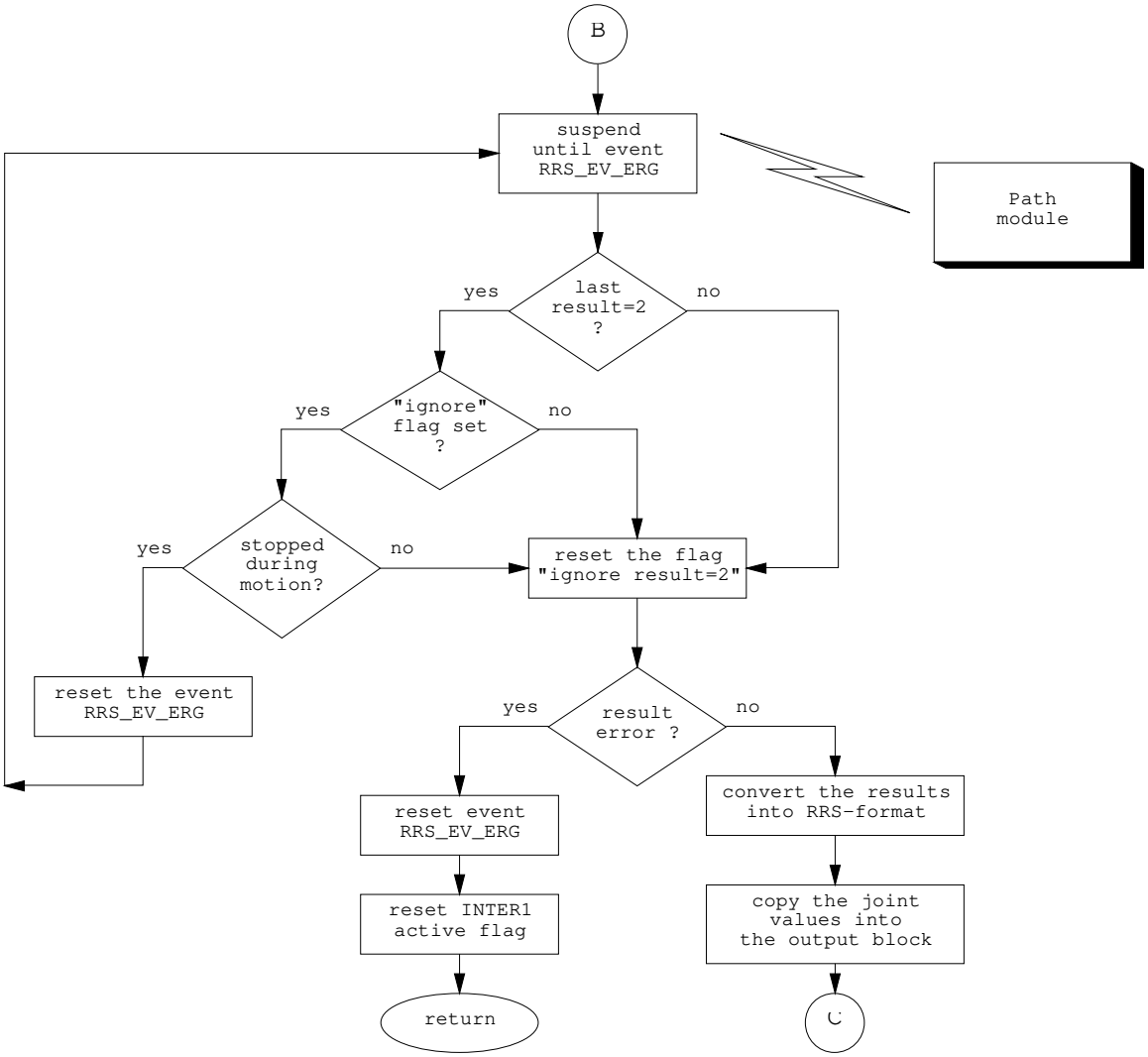


Figure 3.16 : Flow chart of the RRS-Service GET_NEXT_STEP (Part II)

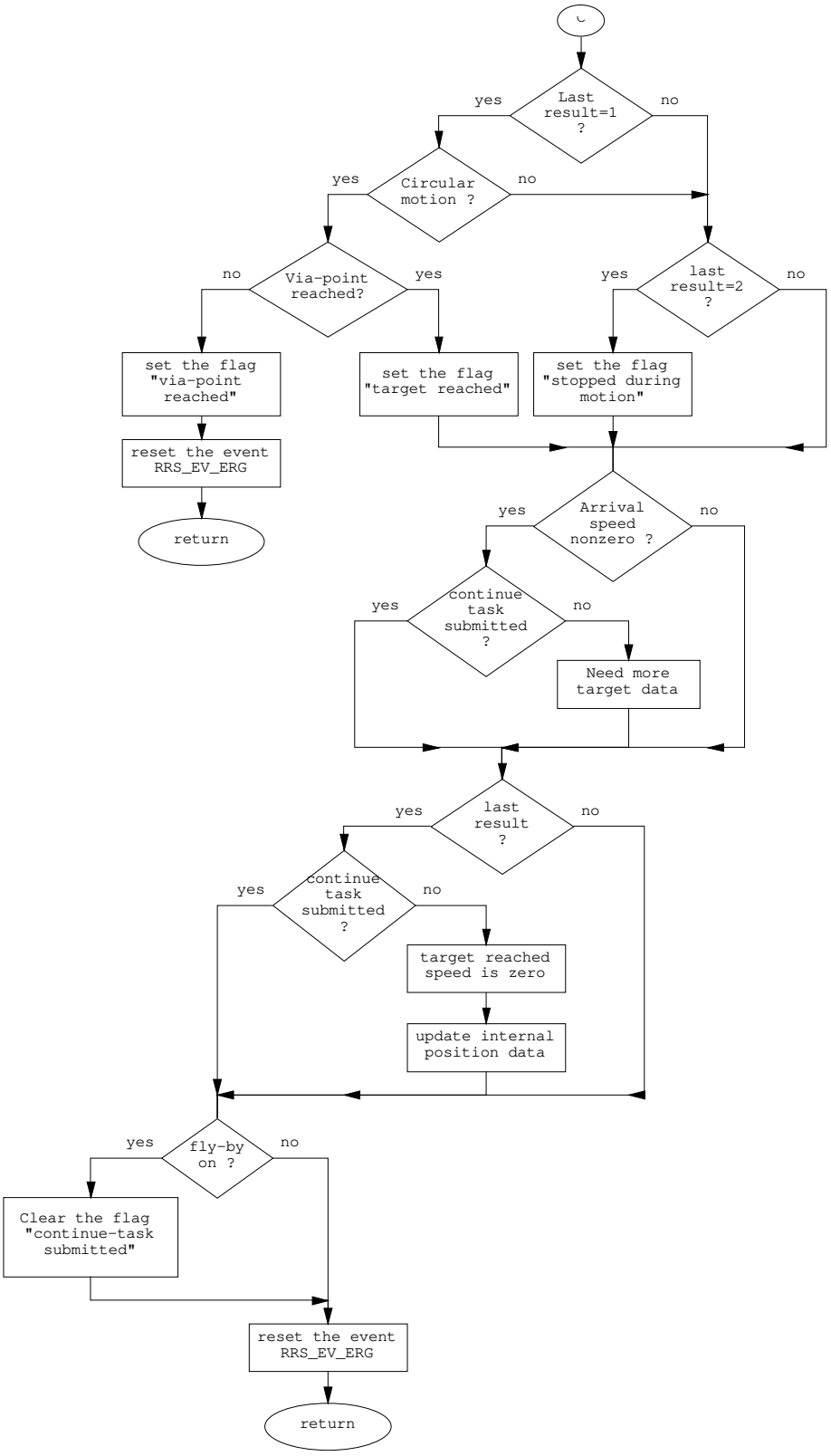


Figure 3.17 : Flow chart of the RRS-Service GET_NEXT_STEP (Part III)

4. Integration of the RCSVW-Module with ROBCAD

4.1 Overview of ROBCAD

The ROBCAD system enables the engineers to perform all of the design work on the screen of a Computer-Aided-Design (CAD) workstation, independent of the workcell on the factory floor. It reduces engineering time, design errors, plant down-time, increases production, throughput, and quality, and let the engineers see with a high degree of visualization the concept working before its actual implementation /ROB/.

By the use of the ROBCAD engineering system, the risk of selecting wrong approaches can be eliminated by quickly modelling, modifying and evaluating various concepts for automating the manufacturing processes and therefore considerable time can be saved. Furthermore, maximum utilization of the equipment is ensured by accurately simulating the operation of the system to optimize component selection, placement, motion, control sequence and cycle time.

By programming the automation system off-line and down-loading the programs generated in ROBCAD to the various device controllers, installation time is reduced and the risk of collision between expensive system components is eliminated. It is also possible to obtain fully-dimensioned hardcopy drawings of individual components or complete workcells.

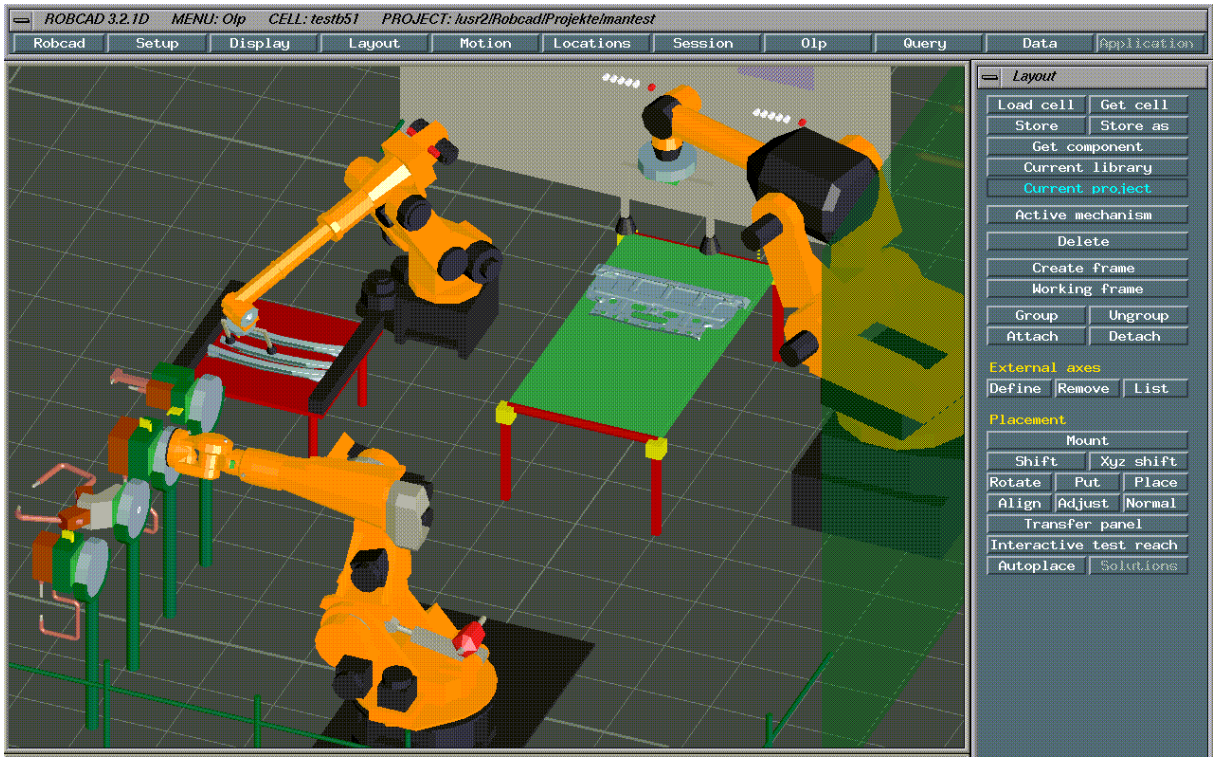


Figure 4.1 : ROBCAD environment

ROBCAD offers a set of tools that allows the user to :

- Create robots and devices with multiple degrees of freedom
- Optimize layouts of the workcells
- Create paths automatically or manually
- Perform dynamic test-reach and automatic placements of the robots under joint limit constraints
- Verify and assure reachability for all desired devices in the workcell
- Write interactively, using a high-level programming language or the robot's own language, programs for each device in the cell
- Test different robot performances of the same tasks
- Check statistically or dynamically, during the design phase or simulation, interferences, for collisions and user defined near misses.
- Measure and improve cycle times
- Download program to robot controller directly or through a post-processor
- Increase the accuracy of the downloaded program by performing dynamic calibration of ROBCAD workcell
- Upload programs for modification and further OLP work

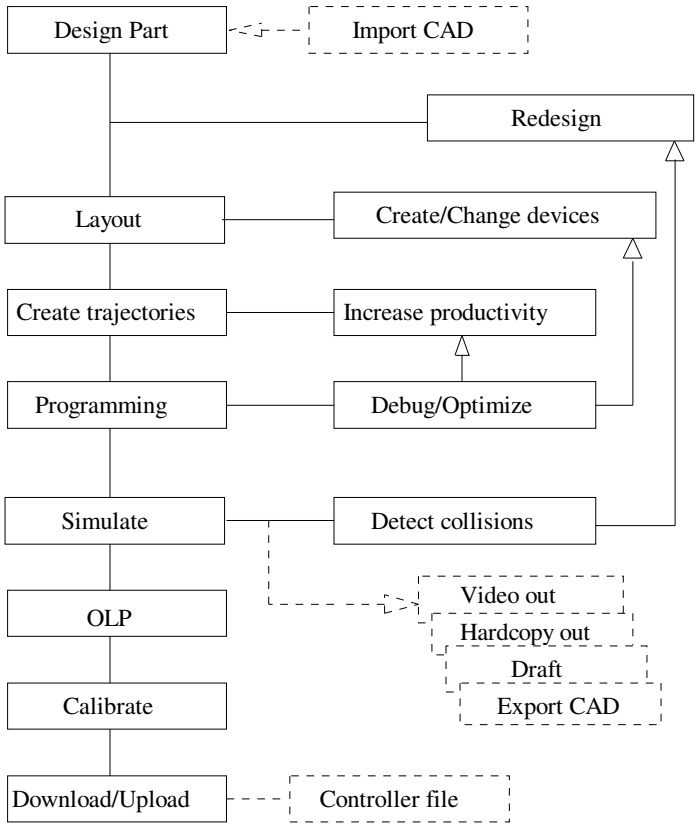


Figure 4.2 : Concept of operation /ROB/

4.2 ROBCAD Off-line Programming (OLP) Development Environment

The OLP Environment is a platform for developing external models of controllers for use within the ROBCAD system. A controller model enables producing programs that the ROBCAD system can simulate, upload, download and edit by means of a teach pendant. The models both generate output files containing controller data, and use data from files to produce actions to be performed at the ROBCAD workstation.

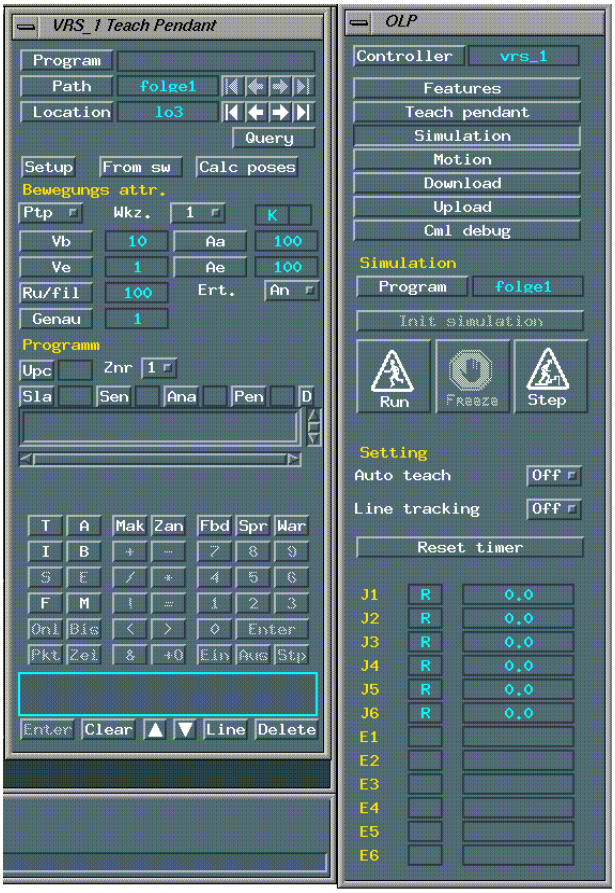


Figure 4.3 : OLP environment

The ROBCAD OLP Development Environment allows developing and maintaining off-line programming packages that are based on the Dynamic Controller Model (DCM) approach. This approach facilitates high-level external controller modeling within ROBCAD by means of user-accessible ASCII files. Using this model does not require compilation or linkage to any ROBCAD applications.

An OLP package includes both control files and action files for

1. Accurately simulating the process. The process is stored in the ROBCAD database as locations, paths, and attributes assigned to the locations and paths.
2. Uploading and downloading the process.
3. Editing the process and adding controller-specific data to it, by means of an attributes-interface application, also referred as teach-pendant.

4.3 The Controller Modelling Language

The Controller Modelling Language (CML) is used in every case that a controller model needs to produce an action in ROBCAD.

The basic element in CML is the statement, of which CML accomodates ten types:

- Motion statements initiate motion of mechanisms
- Set motion parameter statements control motion commands by specifying various motion parameters
- Motion point-definition statements define targets for the next motion statement: target location, via location, etc...
- Set global statements assign new values to global entities such as the TCPF and REFRAME.
- RRS motion statements implement RRS within the OLP development environment
- Control-flow statements enable the CML program to utilize loops and conditions for enhanced efficiency
- Paint statements incorporate painting functions within the CML program
- Synchronization statements synchronize multiple robots for use with the ROBCAD/Session and OLP Application products.
- Message statements send reports to the message window of the application
- Upload statements define sets, paths and locations in ROBCAD, and attach properties and attributes to them. They are also used for workcells and robots.

Control File

ROBCAD makes use of three DCM-based models :

1. *simulation* model effects attribute-based, controller-specific simulation
2. *download* model generates robot native language tasks from sets, paths and locations, together with their generic and controller-specific attributes.
3. *upload* model translates robot native-language tasks to sets, paths and locations.

The approach accomodates a bidirectional interface with external modelling applications. In this model, data is retrieved from the ROBCAD database and is sent to the modelling application. Then, actions are produced in the modelling application and are brought to the ROBCAD system in order to be executed from within ROBCAD. These actions are specified in the Controller Modelling Language (CML). A special control file configures the interface in both directions, by specifying modelling applications for simulation, download and upload, and the data that need to be retrieved as well as its format.

To integrate the RCSVW-Module into ROBCAD, it has been necessary to develop an RRS-oriented action file. This awk-program, which can also be found in the Appendix D, interprets the attributes of work points and generates the necessary RRS-Service calls through available Controller Modelling Language (CML) functions.

The only additional entry in the control file for RRS-Interface based simulation has been the *„motion engine rrs“* statement.

Problems encountered during the integration

CML functions

Once the RRS-oriented action program for simulation has been developed and executed for test paths, it has been observed that some of the CML functions of ROBCAD did not behave as they had been documented.

For instance, the CML function "ExecMotion" should call the RRS-Service GET_NEXT_STEP as long as the latter returns a nonzero "Status" value. However, this function stopped in some cases though the RRS-Service had reported zero as "Status". Another inconsistency which has been noticed with the same function is that it did not always return immediately after receiving Status=1 (by which the RCSVW-Module reports its need for more target data), but continued to call the GET_NEXT_STEP service.

Since neither the original controller software nor its RRS-Interface supports any kind of target buffering, a typical result of the unexpected behaviour mentioned above is that targets are overwritten by newer ones. As errors of this kind occured steadily when using paths consisting of more than three locations, the test motions included in this report have intentionally been limited with three work points, still allowing a comparison of the fly-by behaviours.

Another erroneous point of ROBCAD which has been discovered during the integration is that the unit of angles used for setting initial and target positions was not radian but degrees. In order to be able to perform test motions, the code has been changed for converting the joint values into radian.

System resources

When exiting after an RRS simulation session, ROBCAD did not remove the shared memory segment and the semaphore set of the RRS-Shell from the system. The RRS-Shell had to be extended to make the necessary system calls in order to clear these two IPC identifiers.

5. Conclusion with comparative evaluation of test motions

5.1 Test motions

Before examining some motions which have been performed for test purposes with a KUKA-VK10 type manipulator, an important point about VRS1's functionality and its consequences should be considered : Whatever the type of the programmed motion might be, the VRS1 controller software always needs to be supplied with initial/target positions in joint coordinates. Nevertheless, ROBCAD's general working philosophy is based on locations, whose most important attributes are their position and orientation.

When using the RCSVW-Module with ROBCAD, the locations making up a path have to be converted into poses. This, however, requires the inverse kinematic to be computed for the specific robot being involved. Since the RRS-Service GET_INVERSE_KINEMATIC is not supported, ROBCAD has to compute a pose for each location and then use it with the services SET_INITIAL_POSITION or SET_NEXT_TARGET.

In order to compare ROBCAD's motion engine with the original controller in a reasonable way, one should first ensure that both controllers receive the same joint coordinates as targets. Only if this condition is fulfilled, the joint angle values together with speed profiles may be examined in order to investigate the sources of deviations.

PTP motion

The following test motion has been performed both with the RCSVW-Module and ROBCAD's motion engine.

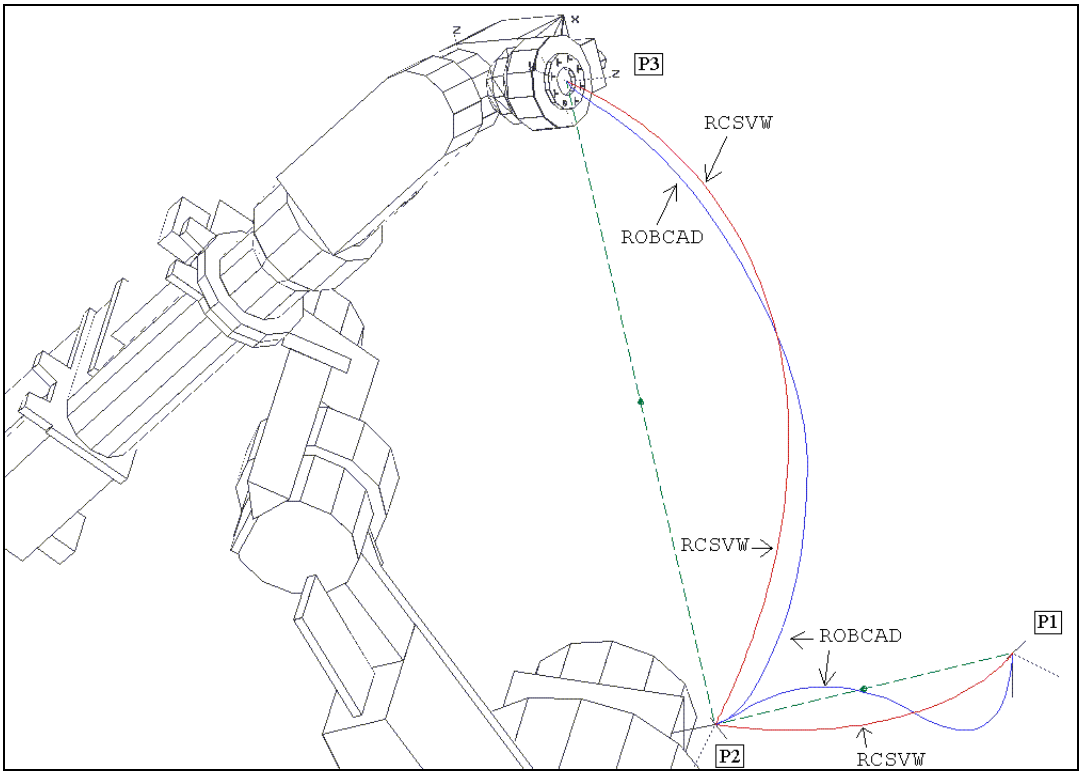


Figure 5.1 : Test motion M_PTP_1

	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
P1	0	0	0	0	0	0
P2	20.5	20.1	-45.4	65.4	63.0	0
P3	-24.6	-26.8	-27.8	30.5	28.5	-150.4

Table 5.1 : Joint angles (in degree) for the motion M_PTP_1

During the motion M_PTP_1, the flyby mode of the controller VRS1 was turned off whereas ROBCAD's flyby zone was selected as *fine*. The acceleration and deceleration values were programmed as 100 % of their maximum values.

Cycle times have been measured for two different speed percentages :

	Speed : 100 %	Speed : 10 %
Real robot	1.60 sec.	6.55 sec.
RRS-Interface	1.59 sec.	6.55 sec.
ROBCAD	1.56 sec.	6.57 sec.

Table 5.2 : Cycle times of the motion M_PTP_1

In this example, as it can also be seen in the pictures taken from various points of view, path deviations up to 55 mm have been noticed.

By means of the speed profiles, one has the possibility to see how each controller computes the joint speeds. In VRS1, PTP-motions are always phase-synchronized. In other words, the speed profiles of all joints have two common points in time, t_a and t_d, which determine the beginning of the acceleration and deceleration phases.

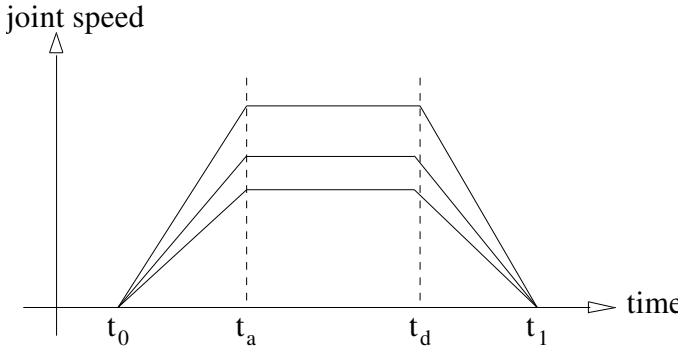
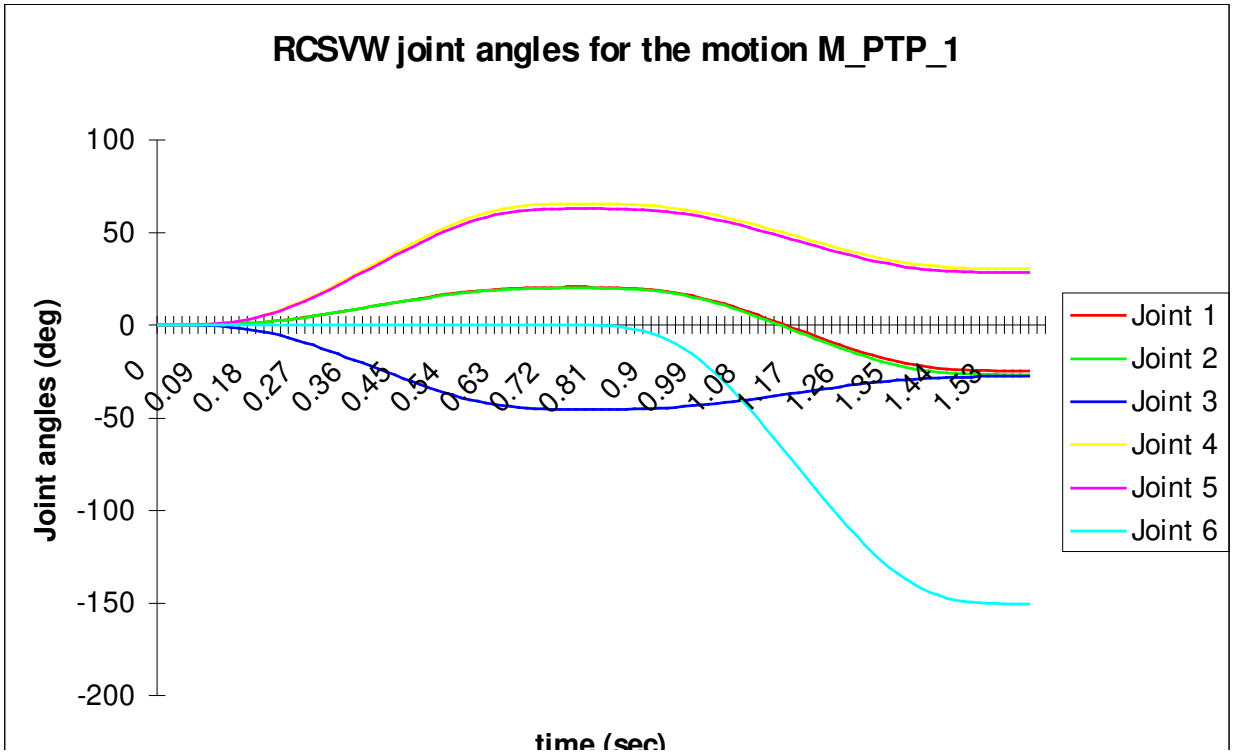
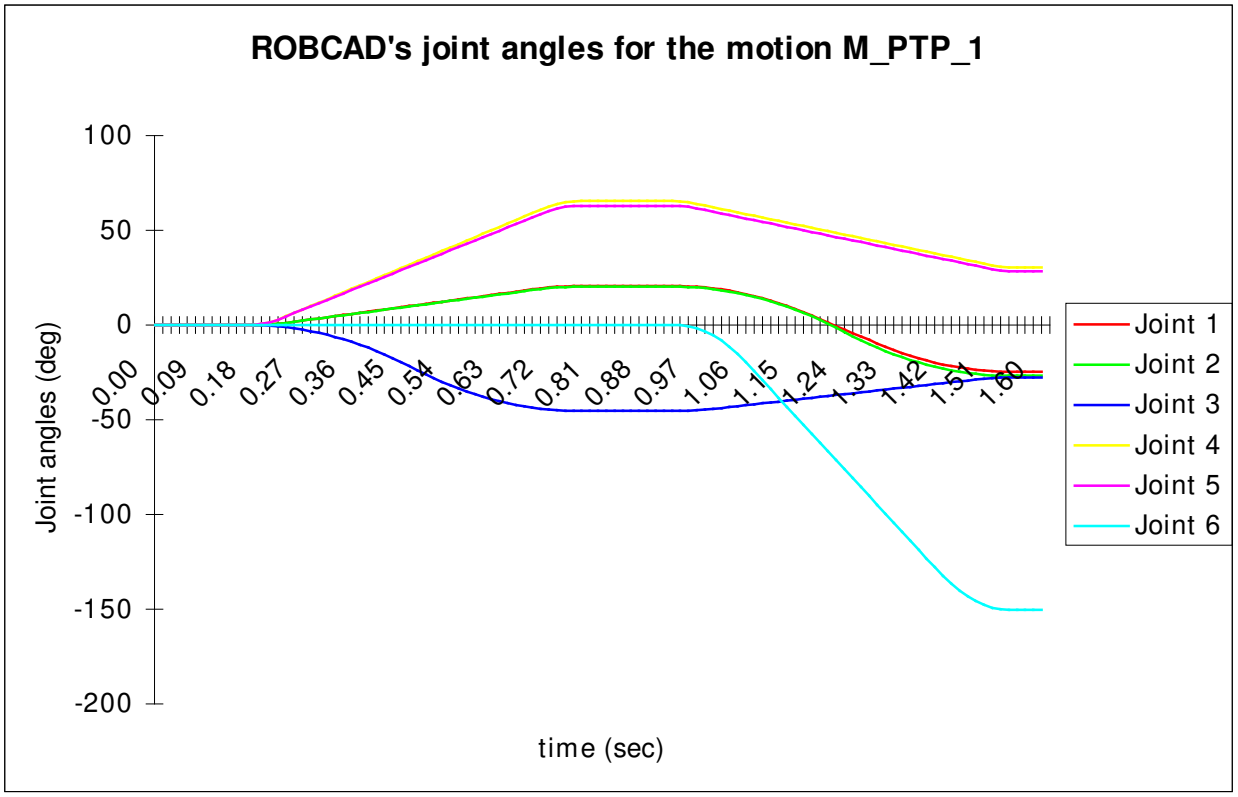


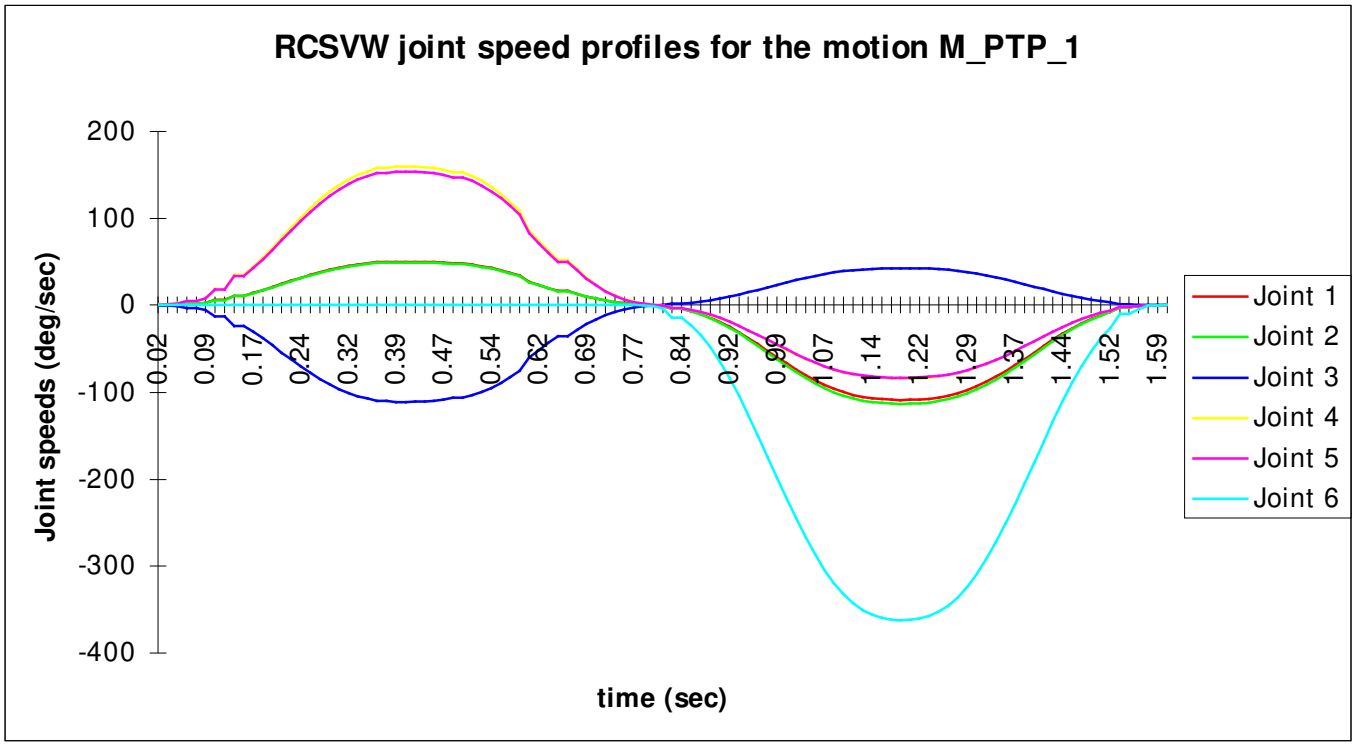
Figure 5.2 : Synchro-PTP speed profile of VRS1

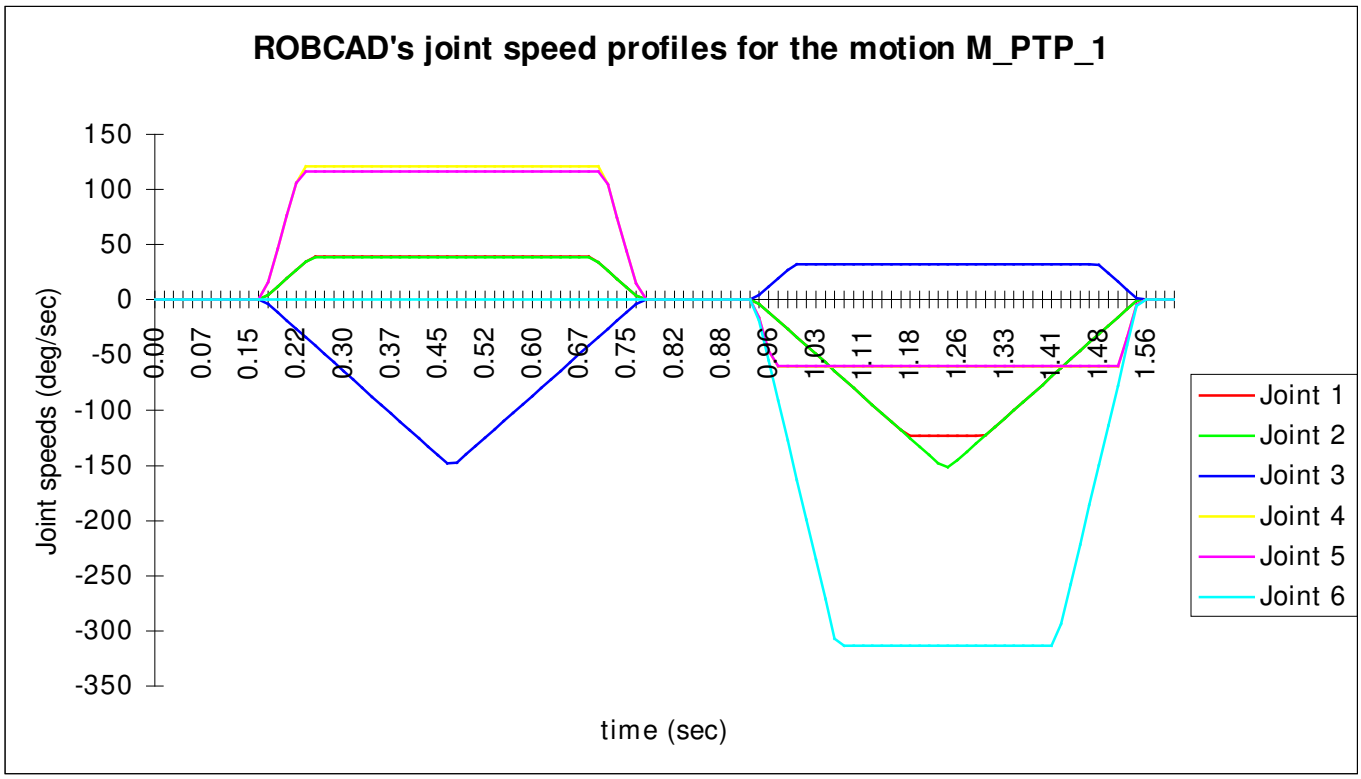
The advantage of such a profile is that stopping and restarting the motion at any point between t_0 and t_1 will not cause any change in the path which will be followed. The uniqueness of the path is resulting from the fact that joints do always achieve the same percentage of their total course at any time t.

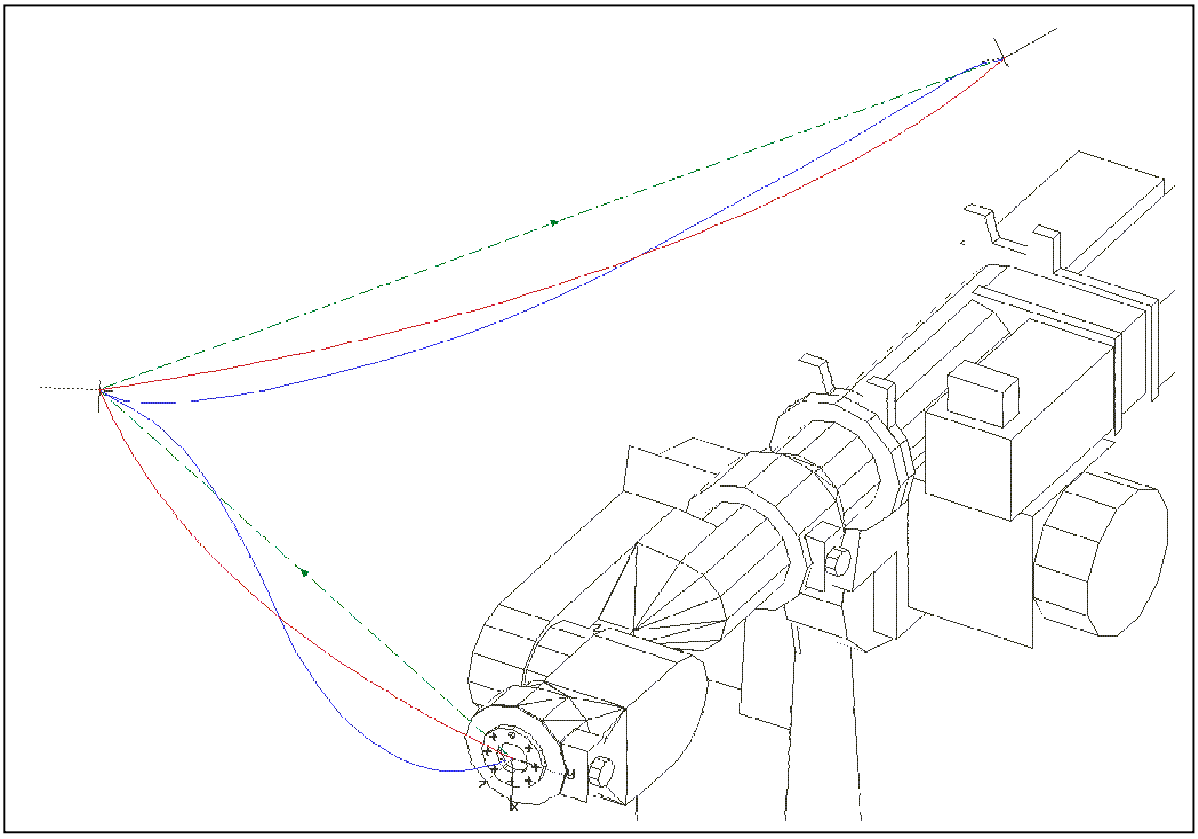
On the other hand, PTP-motions generated by ROBCAD are time-optimal, in the sense that joints do not need to meet the additional conditions by which phase-synchronization is achieved.

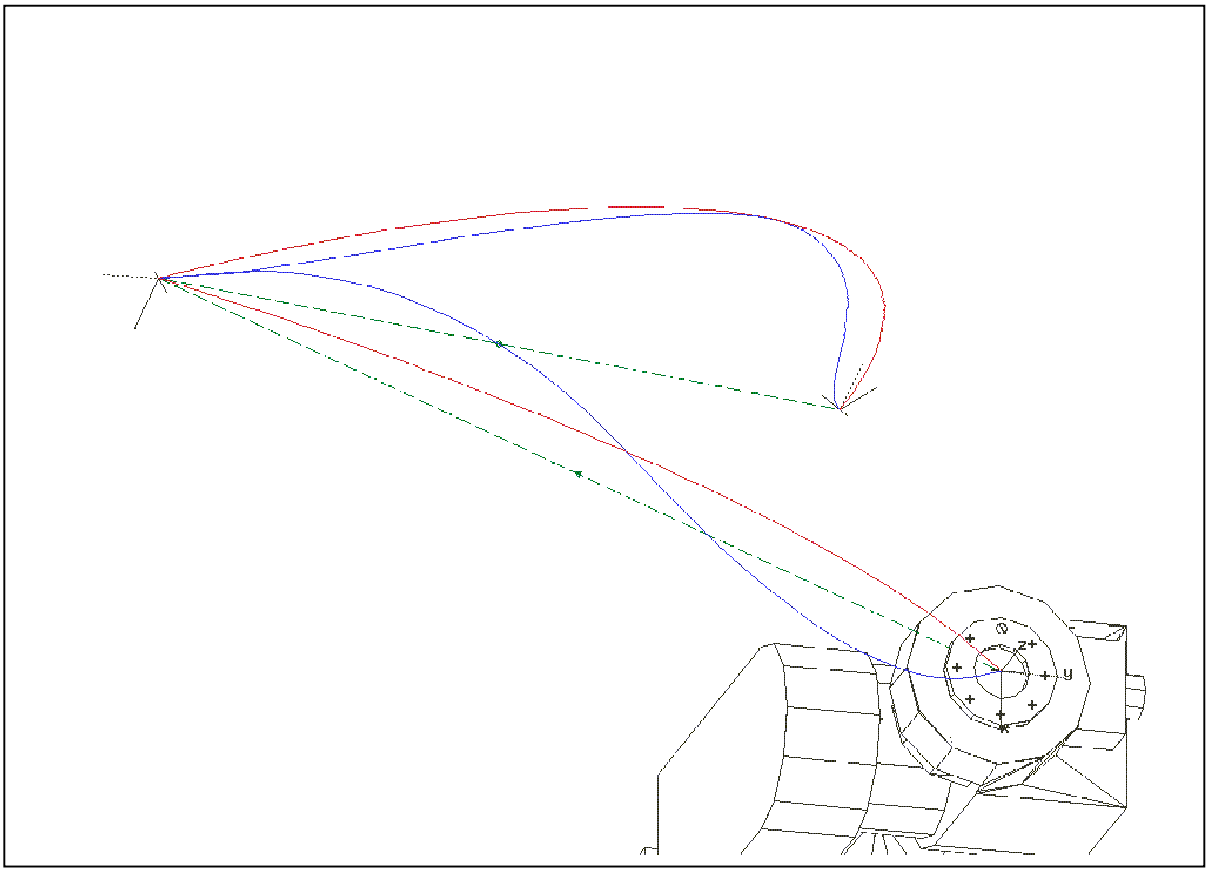

RCSVW joint angles for the motion M_PTP_1
Joint angles (deg)
100
50
0
-50
-100
-150
-200
0
0.09
0.18
0.27
0.36
0.45
0.54
0.63
0.72
0.81
0.9
0.99
1.08
1.17
1.26
1.35
1.44
1.53
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


ROBCAD's joint angles for the motion M_PTP_1
Joint angles (deg)
100
50
0
-50
-100
-150
-200
0.00
0.09
0.18
0.27
0.36
0.45
0.54
0.63
0.72
0.81
0.88
0.97
1.06
1.15
1.24
1.33
1.42
1.51
1.60
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


RCSVW joint speed profiles for the motion M_PTP_1
Joint speeds (deg/sec)
200
100
0
-100
-200
-300
-400
0.02
0.09
0.17
0.24
0.32
0.39
0.47
0.54
0.62
0.69
0.77
0.84
0.92
0.99
1.07
1.14
1.22
1.29
1.37
1.44
1.52
1.59
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


ROBCAD's joint speed profiles for the motion M_PTP_1
Joint speeds (deg/sec)
150
100
50
0
-50
-100
-150
-200
-250
-300
-350
0.00
0.07
0.15
0.22
0.30
0.37
0.45
0.52
0.60
0.67
0.75
0.82
0.88
0.96
1.03
1.11
1.18
1.26
1.33
1.41
1.48
1.56
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6





Linear motion

The joint values used for the test motion M_LIN_1 of linear type are given in the table 5.3.

	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
P4	-53.8	-10.2	27.0	-78.1	-56.6	70.0
P5	-29.7	-34.0	39.4	-80.6	-31.1	80.2
P6	-29.7	-21.9	-0.6	-123.9	-38.2	131.7

Table 5.3 : Joint angles (in degree) for the motion M_LIN_1

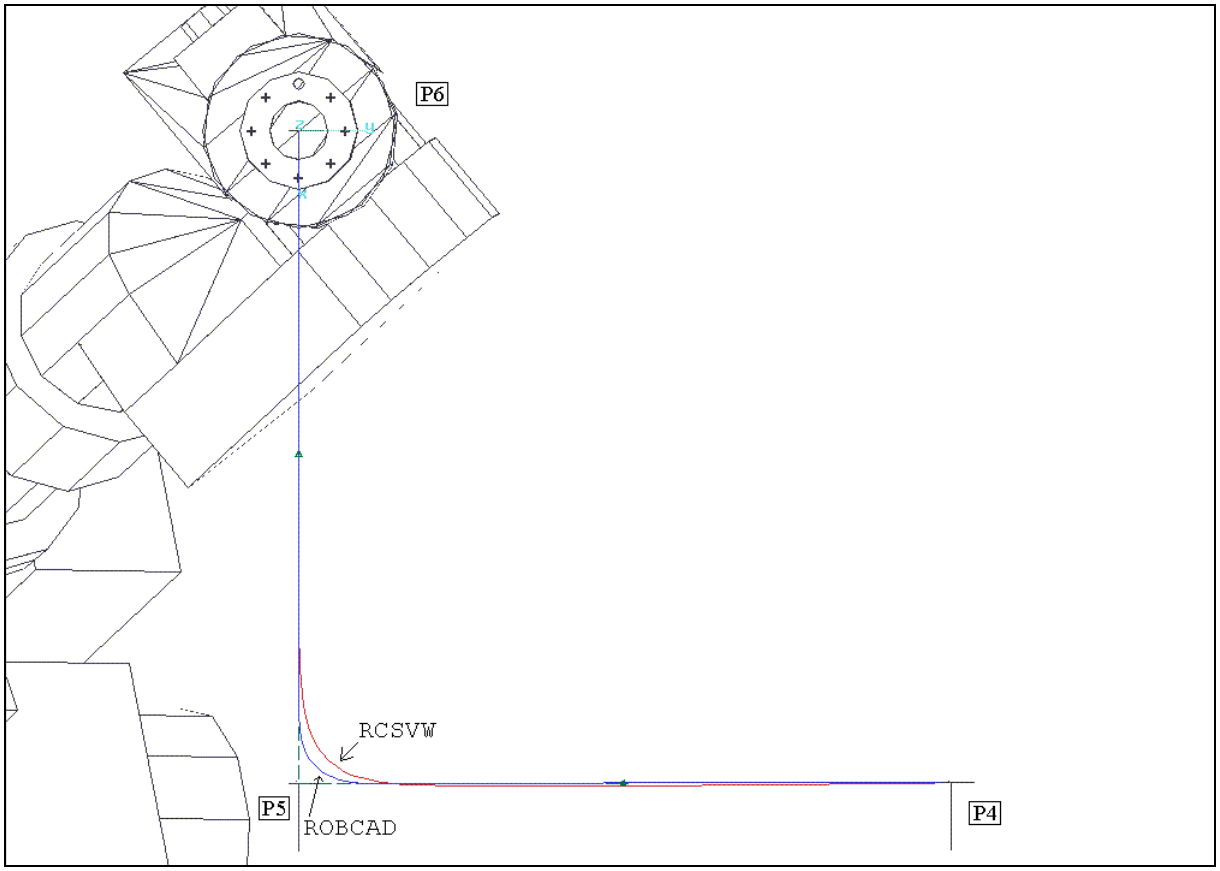
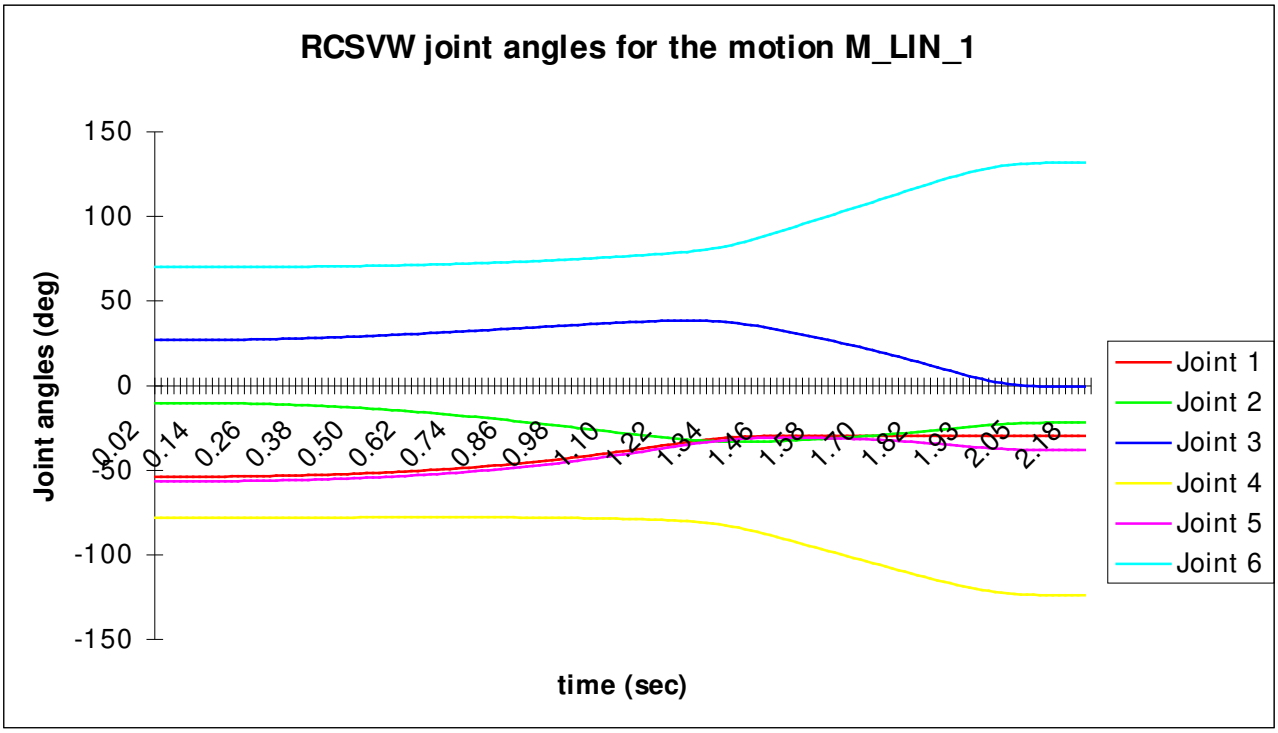
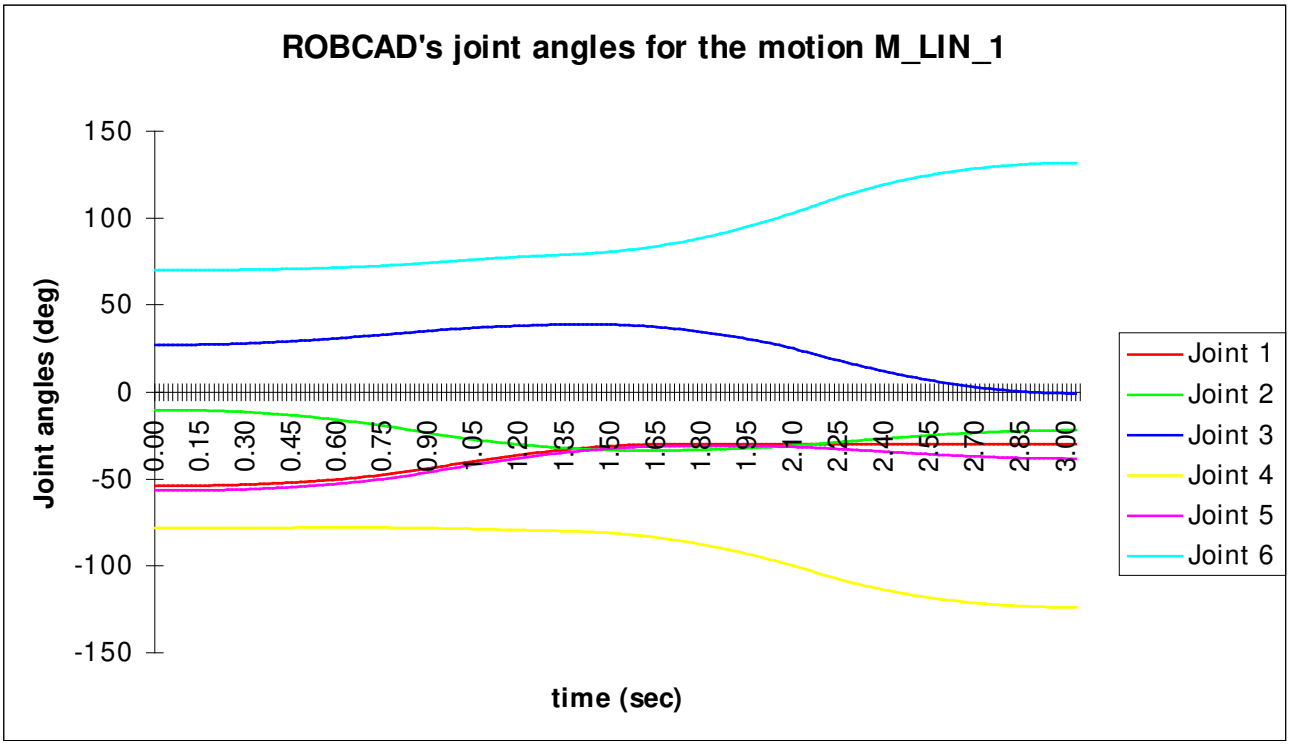
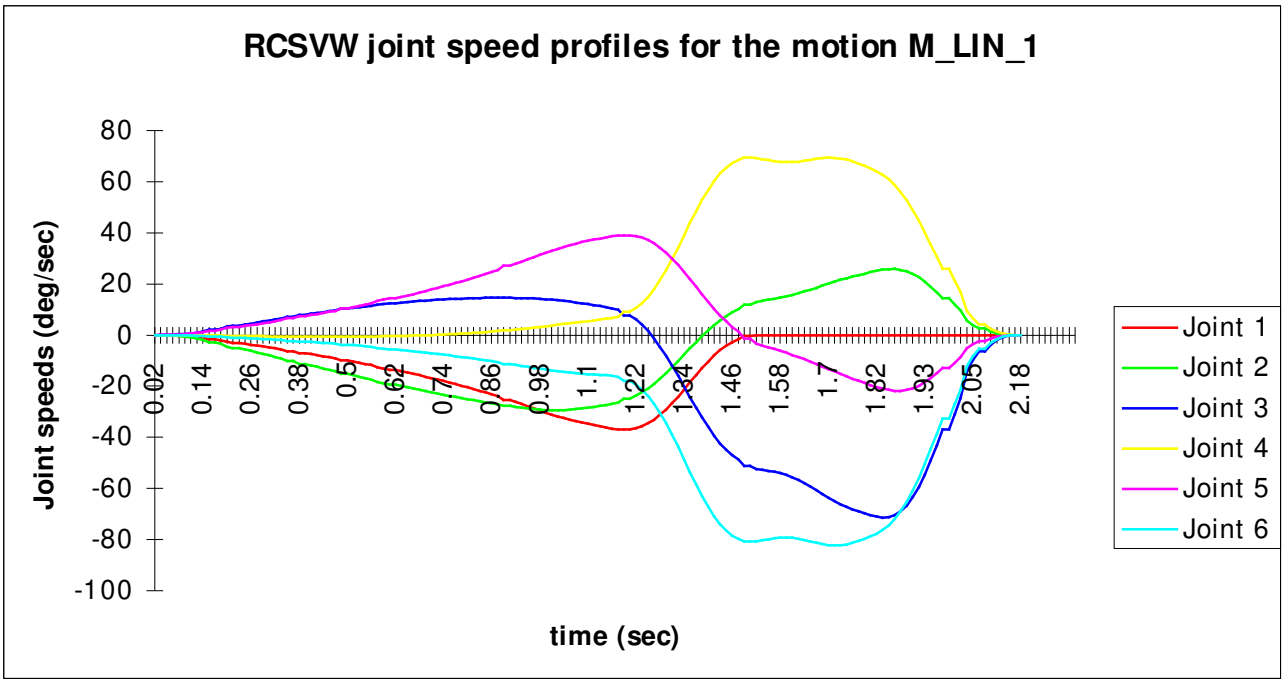


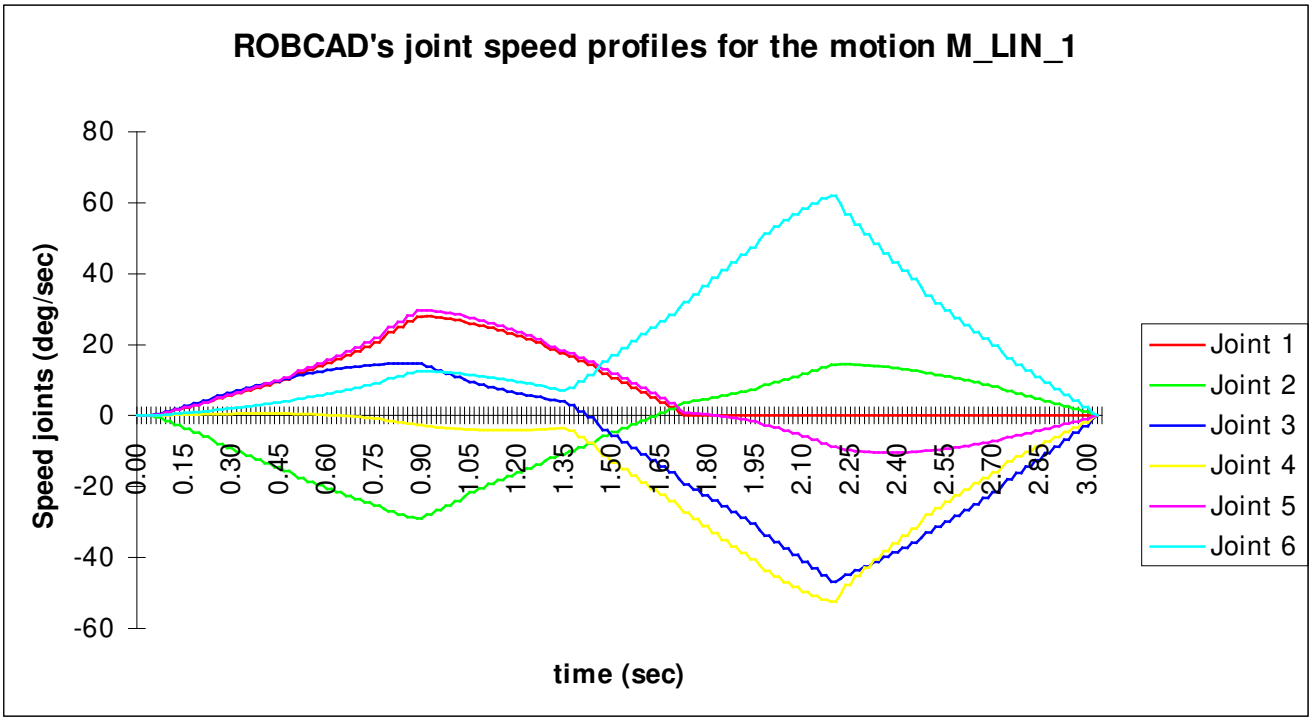
Figure 5.6 : M_LIN_1

During the motion M_LIN_1, the path speed was programmed as 1660 mm/sec with an acceleration/deceleration value of 500 deg/sec^2. The flyby parameter of ROBCAD was selected as „*no deceleration*“, whereas the VRS1 controller used a precision sphere with a radius of 100 mm.


RCSVW joint angles for the motion M_LIN_1
Joint angles (deg)
150
100
50
0
-50
-100
-150
-0.02
0.14
0.26
0.38
0.50
0.62
0.74
0.86
0.98
1.10
1.22
1.34
1.46
1.58
1.70
1.82
1.93
2.05
2.18
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


ROBCAD's joint angles for the motion M_LIN_1
Joint angles (deg)
150
100
50
0
-50
-100
-150
0.00
0.15
0.30
0.45
0.60
0.75
0.90
1.05
1.20
1.35
1.50
1.65
1.80
1.95
2.10
2.25
2.40
2.55
2.70
2.85
3.00
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


RCSVW joint speed profiles for the motion M_LIN_1
Joint speeds (deg/sec)
80
60
40
20
0
-20
-40
-60
-80
-100
0.02
0.14
0.26
0.38
0.5
0.62
0.74
0.86
0.98
1.1
1.22
1.34
1.46
1.58
1.7
1.82
1.93
2.05
2.18
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


ROBCAD's joint speed profiles for the motion M_LIN_1
Speed joints (deg/sec)
80
60
40
20
0
-20
-40
-60
0.00
0.15
0.30
0.45
0.60
0.75
0.90
1.05
1.20
1.35
1.50
1.65
1.80
1.95
2.10
2.25
2.40
2.55
2.70
2.85
3.00
time (sec)
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6

5.2 Implemented RRS-Services

In order to achieve a high degree of RRS-Interface support, the RCSVW-Module has been implemented with a number of functionalities which were not found in the original controller VRS1. The most important extension of this type is the use of a basic kinematic model, allowing the RCSVW-Module to report the TCP in cartesian coordinates as defined in the RRS-Specifications.

Though the current version of the VRS1 controller does not accept targets specified in cartesian coordinates, the necessary data structures together with kinematics-related RRS-Services (GET_ CELL_FRAME, MODIFY_CELL_FRAME, SELECT_WORK_FRAMES) have already been made available.

Table 5.4 shows the RRS-Services which have been implemented and their availability for simulation under the ROBCAD-System. The priorities used in this table have been suggested by some of the project-partners.

RRS Service by name	**Priority**	**ROBCAD support (1)**	**RCSVW support**	**Available for simulation**
INITIALIZE	1	+	+	+
GET CELL FRAME	1	-	+	-
SELECT WORK FRAMES	1	+	+	+
GET INVERSE KINEMATIC	1	+(2)	-	-
SET INITIAL POSITION	1	+	+	+
MODIFY CELL FRAME	1	+	+	+
SET NEXT TARGET	1	+	+	+
GET NEXT STEP	1	+	+	+
TERMINATE	1	+	+	+
GET MESSAGE	1	+	-	-
GET ROBOT STAMP	1	+	+	+
GET FORWARD KINEMATIC	2	+	+	+
MODIFY CELL FRAME	2	+	+	+
CONTROL. POS. TO MATRIX	2	-	+	-
MATRIX TO CONTROL. POS.	2	-	+	-
DEFINE EVENT	2	-	-	-
GET EVENT	2	+	-	-
CANCEL EVENT	2	-	-	-
SELECT FLYBY MODE	2	+	+	+
CANCEL FLYBY CRITERIA	2	+	-	-
SELECT FLYBY CRITERIA	2	+	-	-
SET FLYBY CRITERIA PARA	2	+	+	+
SELECT POINT ACCURACY	2	+	-	-
SET POINT ACCURACY PAR	2	+	-	-
SELECT MOTION TYPE	2	+	+	+
SET CART. ORIENT. ACCEL.	2	+	-	-

RRS Service by name	Priority	ROBCAD support [1]	RCSVW support	Available for simulation
SET CART. ORIENT. SPEED	2	+	-	-
SET CART. POS. ACCEL.	2	+	+	+
SET CART. POSITION SPEED	2	+	+	+
SET INTERPOLATION TIME	2	+	-	-
SET JOINT ACCEL.	2	+	-	-
SET JOINT SPEEDS	2	+	+	+
SET MOTION FILTER	2	-	-	-
SET JOINT JERK	2	-	+	-
RESET	2	-	+	-
DEBUG	2	+	+	+
STOP MOTION	3	-	+	-[3]
CANCEL MOTION	3	+	+	-[3]
CONTINUE MOTION	3	+	+	-[3]
SELECT TARGET TYPE	3	-	-	-
SELECT ORI. INTERP. MODE	3	+	-	-
SELECT DOMINANT INTER.	3	-	-	-
SELECT TRACKING	3	-	-	-
SELECT TRAJECT. MODE	3	+	-	-
SET CONVEYOR POSITION	3	-	-	-
SET OVERRIDE POSITION	3	+	+	-[3]
SET OVERRIDE SPEED	3	+	+	-[3]
SET OVERRIDE ACCEL	3	+	+	-[3]
SELECT WEAVING GROUP	3	-	+	-
SELECT WEAVING MODE	3	-	+	-
SET WEAVING GROUP PAR	3	-	+	-
SET PAYLOAD PARAMETER	3	-	-	-
SET CONFIG. CONTROL	3	-	-	-
SET CURRENT TARGET ID	3	-	-	-
REVERSE MOTION	4	-	-	-
LOAD RCS DATA	4	-	-	-
SAVE RCS DATA	4	-	-	-
GET RCS DATA	4	-	-	-
MODIFY RCS DATA	4	-	-	-

Table 5.4 : Table of RRS-Services implemented for VRS1

[1] ROBCAD V 3.2.1, basing on the OLP Development Environment Reference Manual V3.2.1.
[2] Used only by the „RRS Test Application“ of ROBCAD.
[3] This service can not be used yet due to the lack of ROBCAD’s RRS support.

5.3 Summary of the work

The main objective of this work has been to develop the RRS-Interface for the VRS1 Robot Controller and to come up with a working RCS-Module under ROBCAD. After six months of intensive programming, it has been ultimately possible to achieve this goal and evaluate the first simulation results where the original controller's behaviour could be compared with that of ROBCAD's motion planner.

Problems with ROBCAD's RRS-Interface

Regrettably, the RRS capabilities of the ROBCAD-System proved to be unsatisfactory, in the sense that a number of inconsistencies encountered throughout the project were finally accepted as „already recognized problems" by the company Tecnomatix Technologies.

In some cases, it has been possible to re-program parts of the RCSVW-Module to enable the execution of test motions (conversion of joint angles, removal of the IPC facilities at exit), whereas in some others, severe restrictions (number of locations on a programmed path) did not allow the simulation of real processes extracted from the manufacturing environment.

The future of the RCSVW-Module in the Volkswagen Group

The Volkswagen Group is planning to bring the RCSVW-Module into industrial use beginning from the second quarter of 1996. The first off-line programming tasks will include the planning of the production lines for new car models such as the B5 PASSAT, A4 GOLF and A6 AUDI.

With the use of the RRS-Interface in off-line programming systems, the current simulation accuracy of 95 % is expected to raise to 99 %. Since the manufacturing processes mentioned above will involve the off-line programming of thousands of industrial robots, even the short-term benefits of such an improvement can be already considered of economic significance.

Furthermore, the availability of the original path control routines of VRS1 as an independent RCSVW-Module enables the portability of this very accurate simulation package among other CAR-Tools supporting the RRS-Interface. From this point of view, the RCSVW-Module can be also regarded as a simulation product, deliverable as black-box to customers desiring to make use of the state-of-the-art simulation tools.

References :

/BAC86/ M. Bach „Design of the UNIX Operating System“ Prentice-Hall, 1986

/BEC92/ J.Bechtloff „Interpolationsverfahren höheren Grades für Robotersteuerungen“ Braunschweiger Schriften zur Mechanik, TU-Braunschweig, 1992

/BEN90/ Ben-Ari "Principles of Concurrent and Distributed Programming" Prentice Hall International Series in Computer Science, 1990

/BER94/ R.Bernhardt, G.Schreck, C.Willnow „Accuracy Enhancement In Off-line Programming For Industrial Robots“ presented in „Next Steps For Industrial Robotics“, May 1994

/BERN94/ R.Bernhardt, G.Schreck, C.Willnow „Time and Cost Saving by Using Accurate Simulation Tools for Planning Manufacturing Systems with Robots“, Workshop on Robotics, Industrial Robots Automatic Handling and Assembly Equipment in Present and Future Manufacturing Systems, Budapest ‘94.

/BER95/ R.Bernhardt, G.Schreck, C.Willnow „Deviation of Simulation and Reality in Robotics : Causes and Counter Measures“, International Symposium on Automotive Technology and Automation“, September 1995, Germany

/BES95/ P. Beske „Programmieren von Robotern, Roboterverkettung und Zellenlayout“ Volkswagen, Elektroplanung Abt. 1995

/BOY89/ N.P.Boysen „A Robot Controller Concept for Sensory Controlled Motion“ Technical University of Denmark, June 1989

/DAN91/ Danneger "Parallele Prozesse unter UNIX : Simulation und Anwendung bekannter Synchronisationsmethoden in C unter UNIX", Hanser Programmtexte, 1991

/HOC90/ C. Hockemeyer „Implementation von Spiegelung und Transformation in der Robotersteuerung VRS1“, Studienarbeit, TU-Braunschweig.

/JAN91/ D. Janssen „Implementierung Online-fähiger Spline-Interpolationsalgorithmen in die VW-Bahnsteuerung VRS1“, Diplomarbeit, TU-Braunschweig.

/PAU81/ R. Paul, „Robot Manipulators: Mathematics, Programming and Control“ MIT Press, 1981

/ROB/ ROBCAD Technical Description
Tecnomatix Technologies Ltd., Israel

/RRS95/ Realistic Robot Simulation Interface Specifications, Version 1.1, 1995

/STE92/ W.R.Stevens „Advanced UNIX Programming in the UNIX Environment“
Addison-Wesley Professional Computing Series, 1992

/VRS1/ Volkswagen-Roboter-Steuerung (VRS) 1 Programmier-handbuch
Volkswagen, March 1994.

/VME89/ VMEPROM Version 2, Users Manual
Force Computers, June 1989

Other references :

J.Peek, T.O'Reilly, M.Loukides
„UNIX Power Tools“
O'Reilly & Associates, 1994

H.Heroldt
"UNIX und seine Werkzeuge : awk und sed"
Addison Wesley, 1994

IBM RT Advanced Interactive Executive Operating System
AIX Technical Reference : System Calls and Subroutines
Programming Tools and Interfaces, Version 2.2, 1988

IBM AIX Version 3.2 for RISC System/6000
XL C User's guide, XL C Language Reference, 1991

Control Data, Cyber 910
Programmer's Reference Manual, Programmer's Guide, 1990

Appendices

A. Further test motions for comparison with ROBCAD
B. Exemplar log entries of simulation
C. Complete list of RRS-Services
D. Control file and RRS-oriented action program used with ROBCAD
E. Source code of the RCSVW-Module with compilation directives

APPENDIX A : Further test motions for the comparison of the RCSVW-Module with ROBCAD

Below are a number of simulation results where ROBCAD's motion engine is being compared with the RCSVW-Module.

Table A.1 shows the parameters which have been used during these motions and tables A.2 through A.6 show the positions of the programmed locations in joint coordinates.

	M_PTP_2	**M_PTP_3**	**M_PTP_4**	**M_PTP_5**	**M_LIN_2**
Motion type	PTP	PTP	PTP	PTP	Linear
Speed	100 %	100 %	100 %	100 %	1660 mm/sec
Fly-by mode	off	off	off	off	on
Maximum path deviation	40 mm	90 mm		30 mm	10 mm fly-by : 25 mm

Table A.1 : Table of test motions performed without speed analysis

M_PTP_2

	Joint 1	**Joint 2**	**Joint 3**	**Joint 4**	**Joint 5**	**Joint 6**
P7	0.0	0.0	0.0	0.0	0.0	0.0
P8	0.0	-3.0	-75.0	-76.6	-65.2	0.0

Table A.2 : Joint angles (in degree) of the motion M_PTP_2

M_PTP_3

	Joint 1	**Joint 2**	**Joint 3**	**Joint 4**	**Joint 5**	**Joint 6**
P9	0.0	0.0	0.0	0.0	0.0	0.0
P10	13.2	-27.2	-27.2	87.2	-89.4	0.0

Table A.3 : Joint angles (in degree) of the motion M_PTP_3

M_PTP_4

	Joint 1	**Joint 2**	**Joint 3**	**Joint 4**	**Joint 5**	**Joint 6**
P14	0.0	0.0	0.0	0.0	0.0	0.0
P15	33.6	36.9	2.1	-7.7	-81.4	41.6

Table A.4 : Joint angles (in degree) of the motion M_PTP_4

M_PTP_5

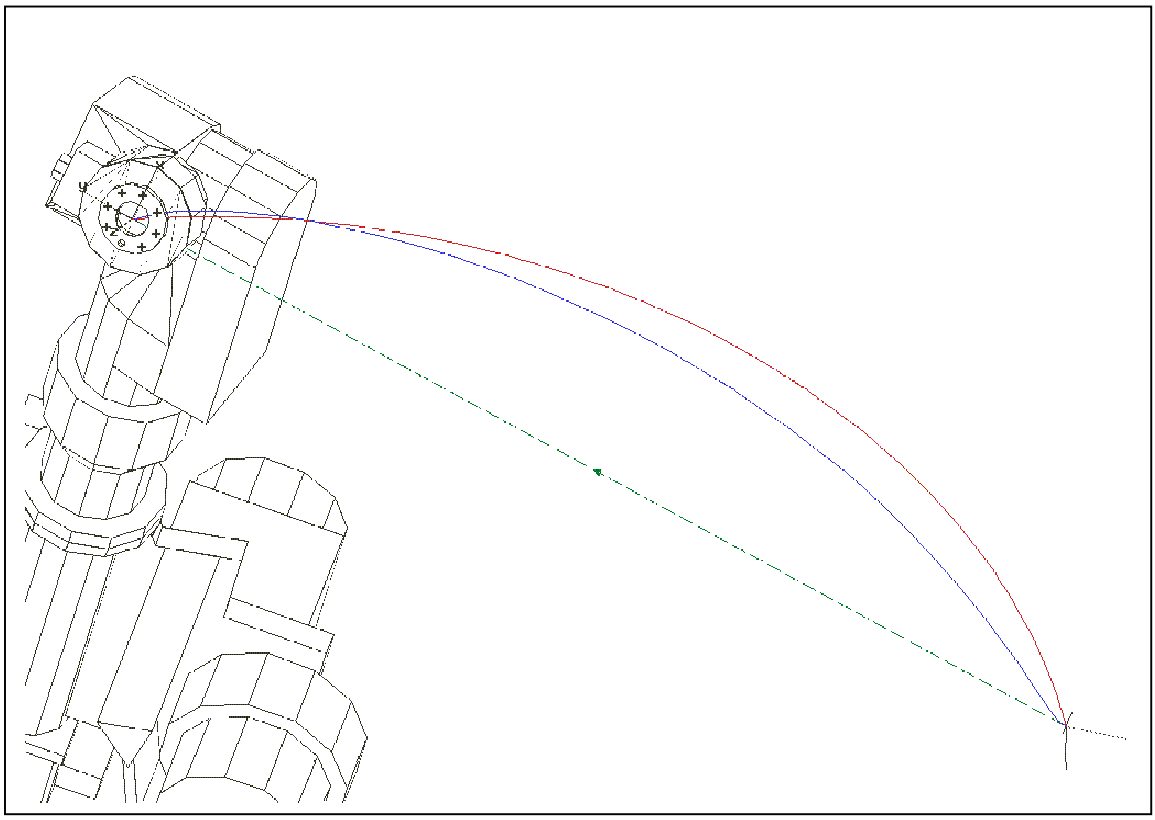
	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
P16	0.0	0.0	0.0	0.0	0.0	0.0
P17	16,5	0.0	0.0	0.0	-50.0	0.0
P18	25.0	-36.8	-26.3	-52.9	47.5	-17.8

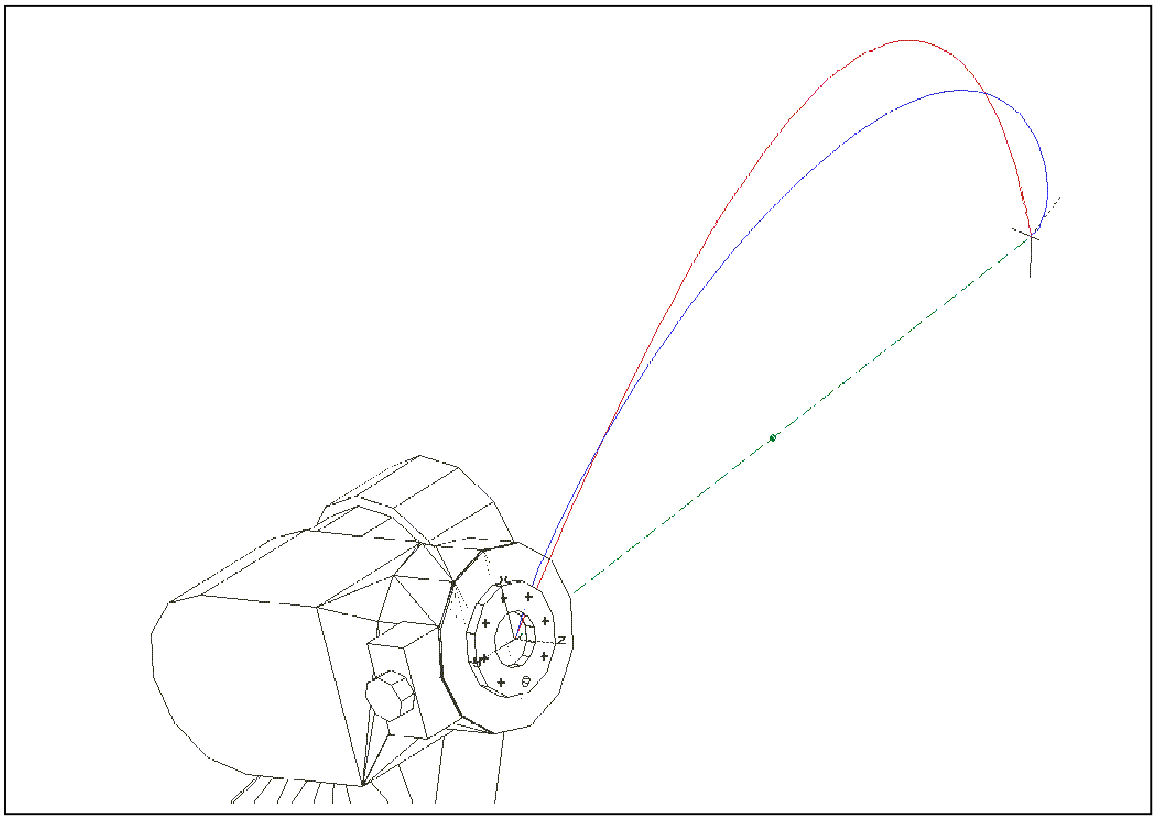
Table A.5 : Joint angles (in degree) of the motion M_PTP_5

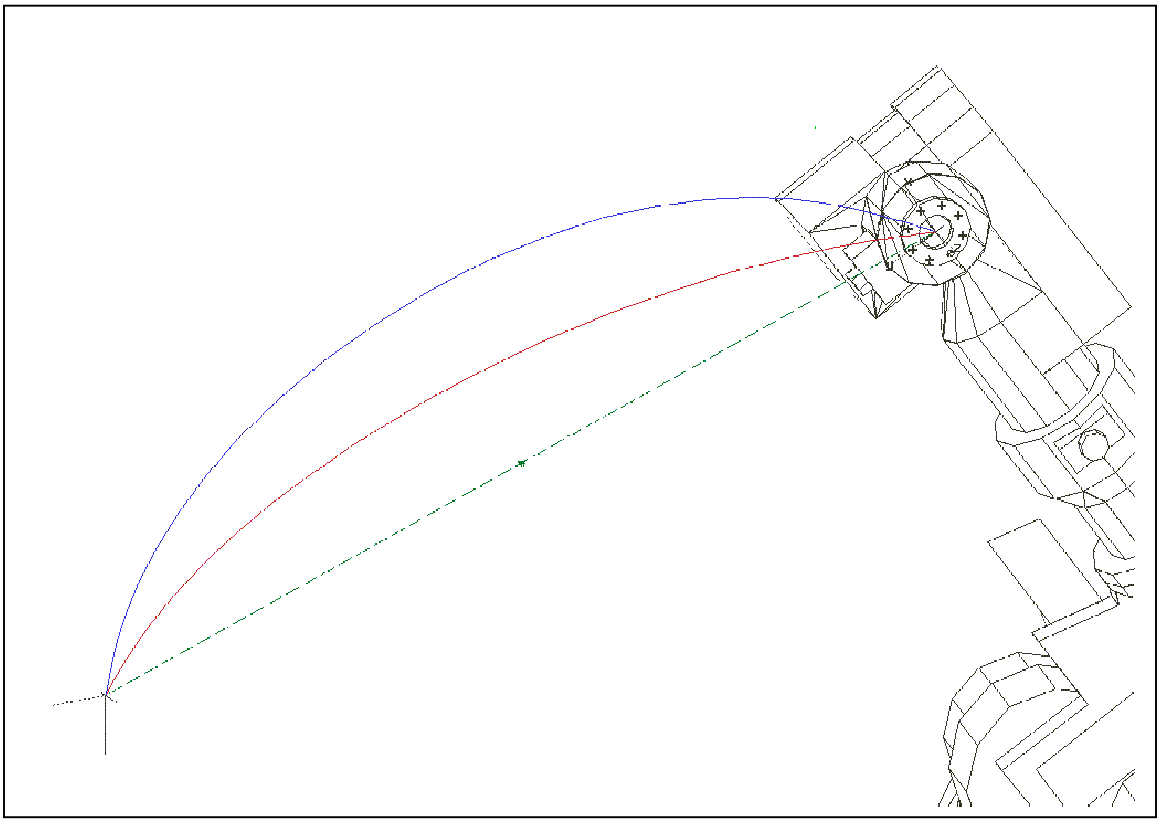
M_LIN_2

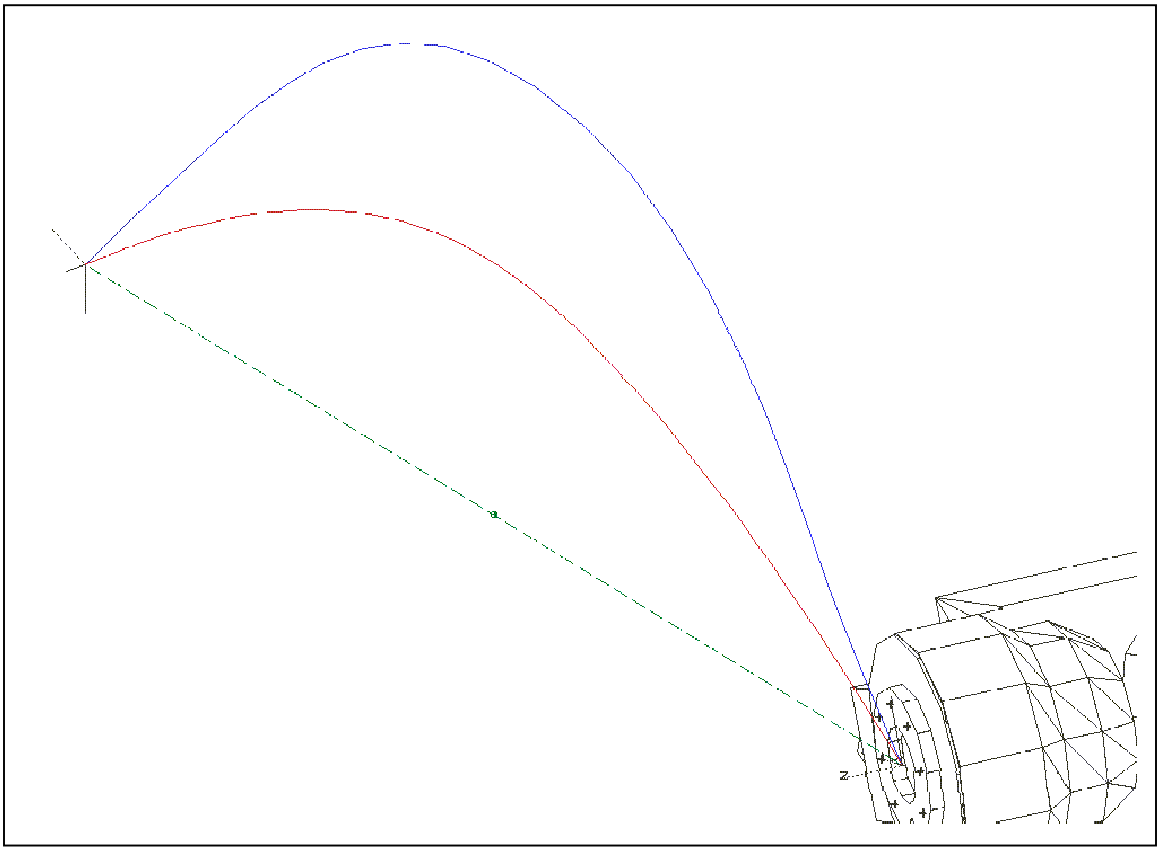
	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
P11	-4.5	-19.4	42.2	-11.4	-23.3	-169.7
P12	23.1	35.5	4.3	-74.7	-73.4	-312.1
P13	60.9	50.4	30.9	-29.4	-82.8	3.2

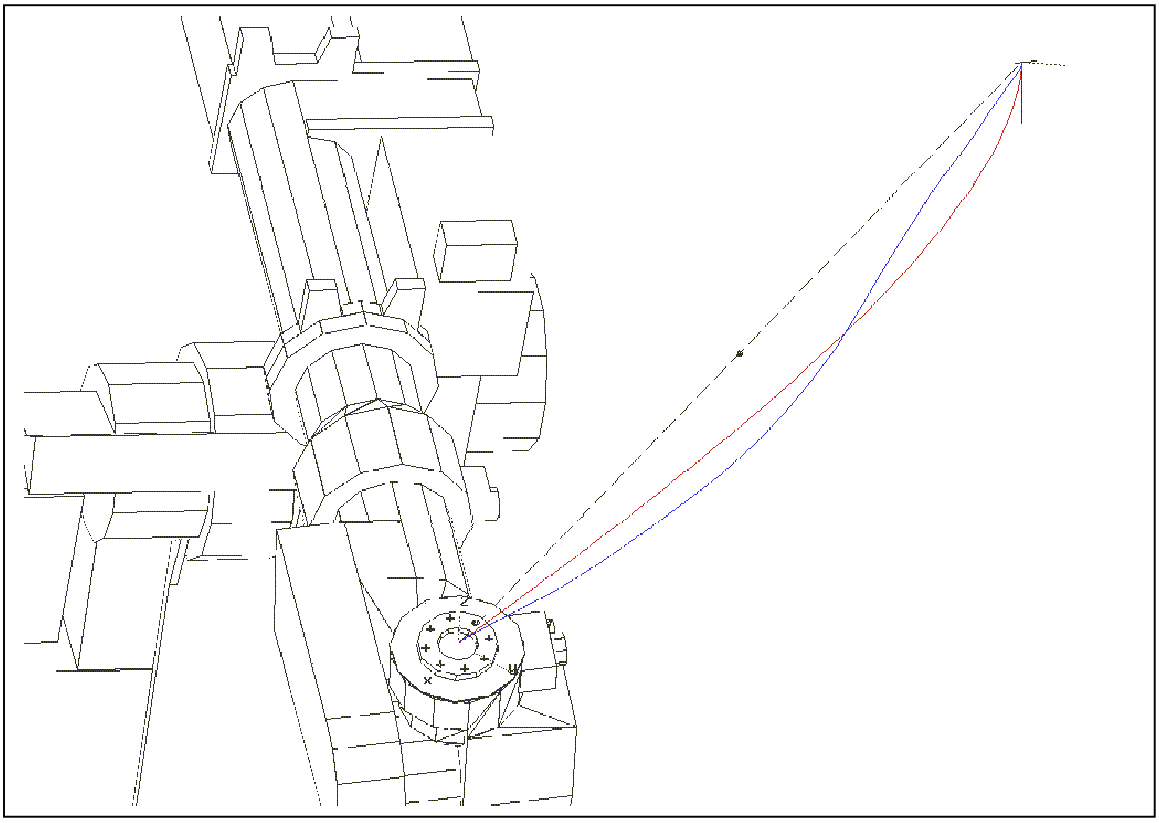
Table A.6 : Joint angles (in degree) of the motion M_LIN_2

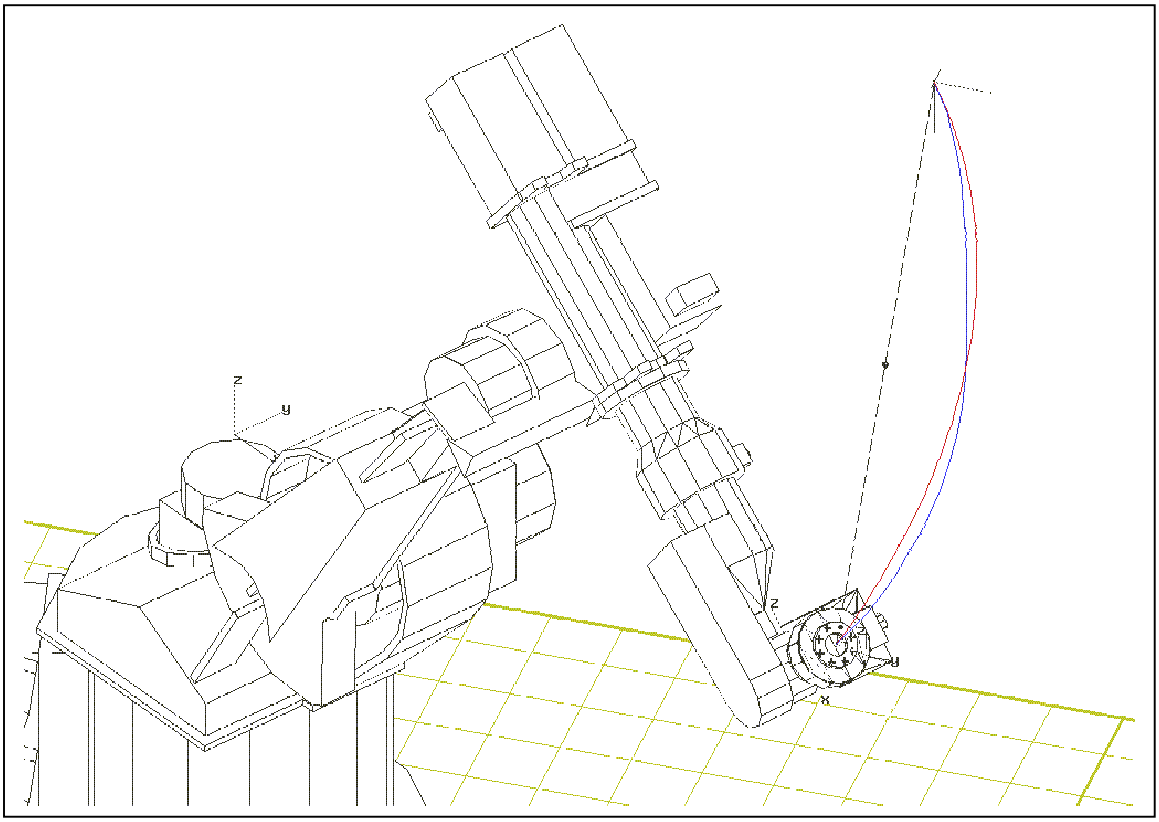


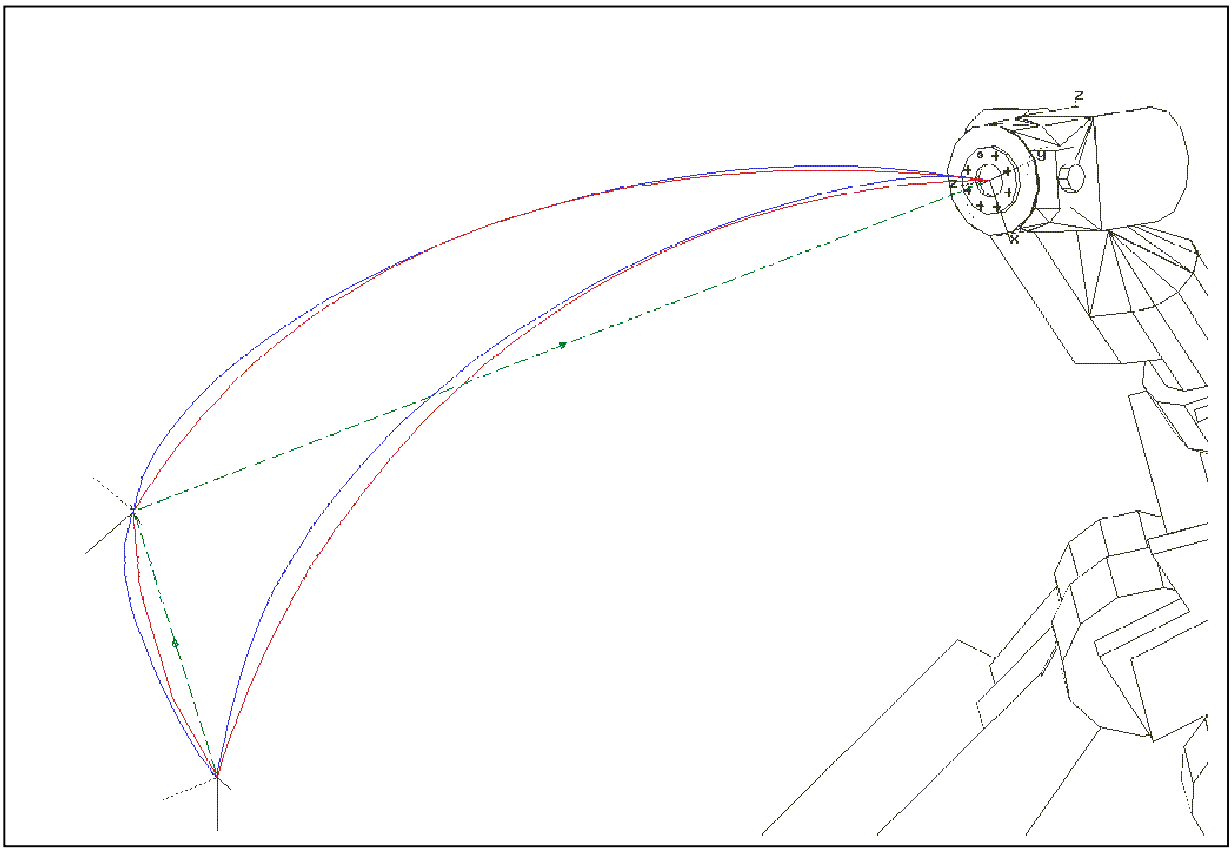


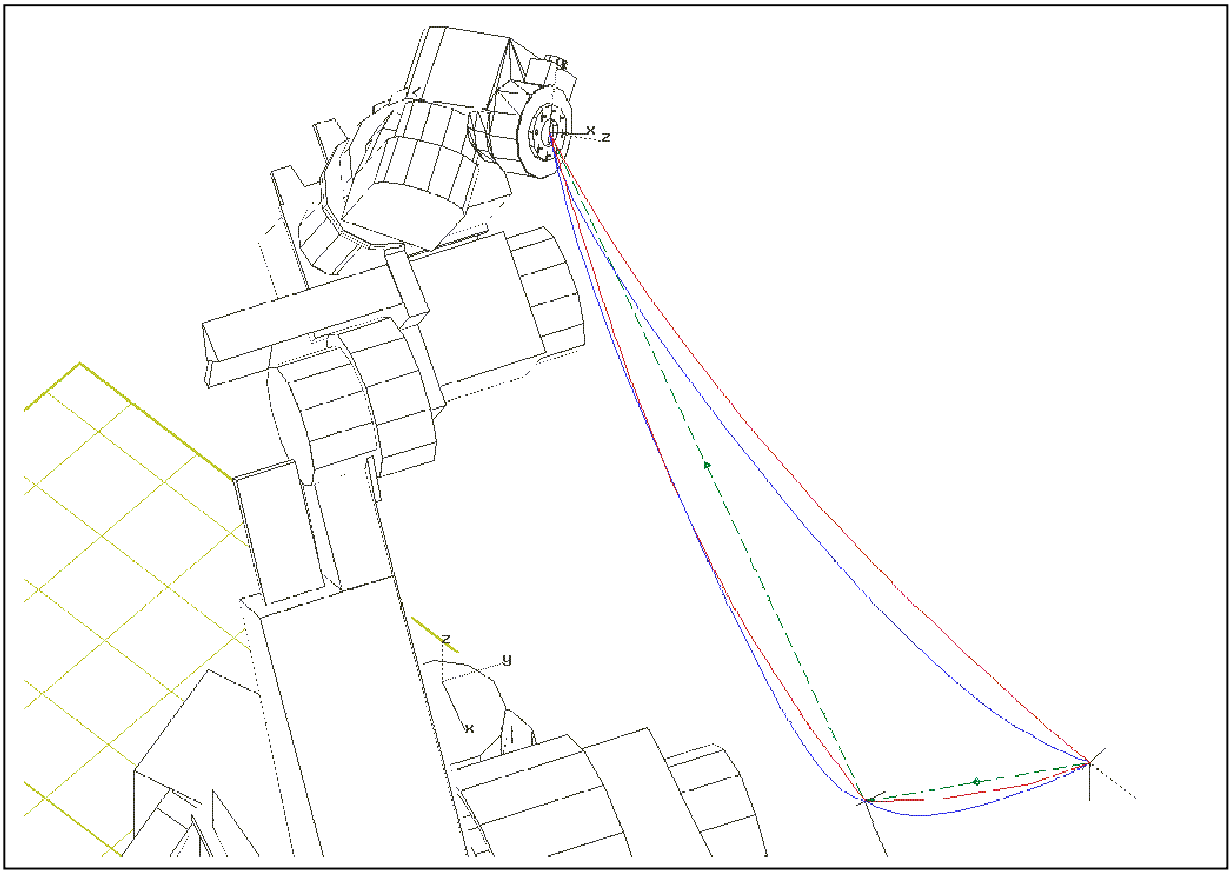


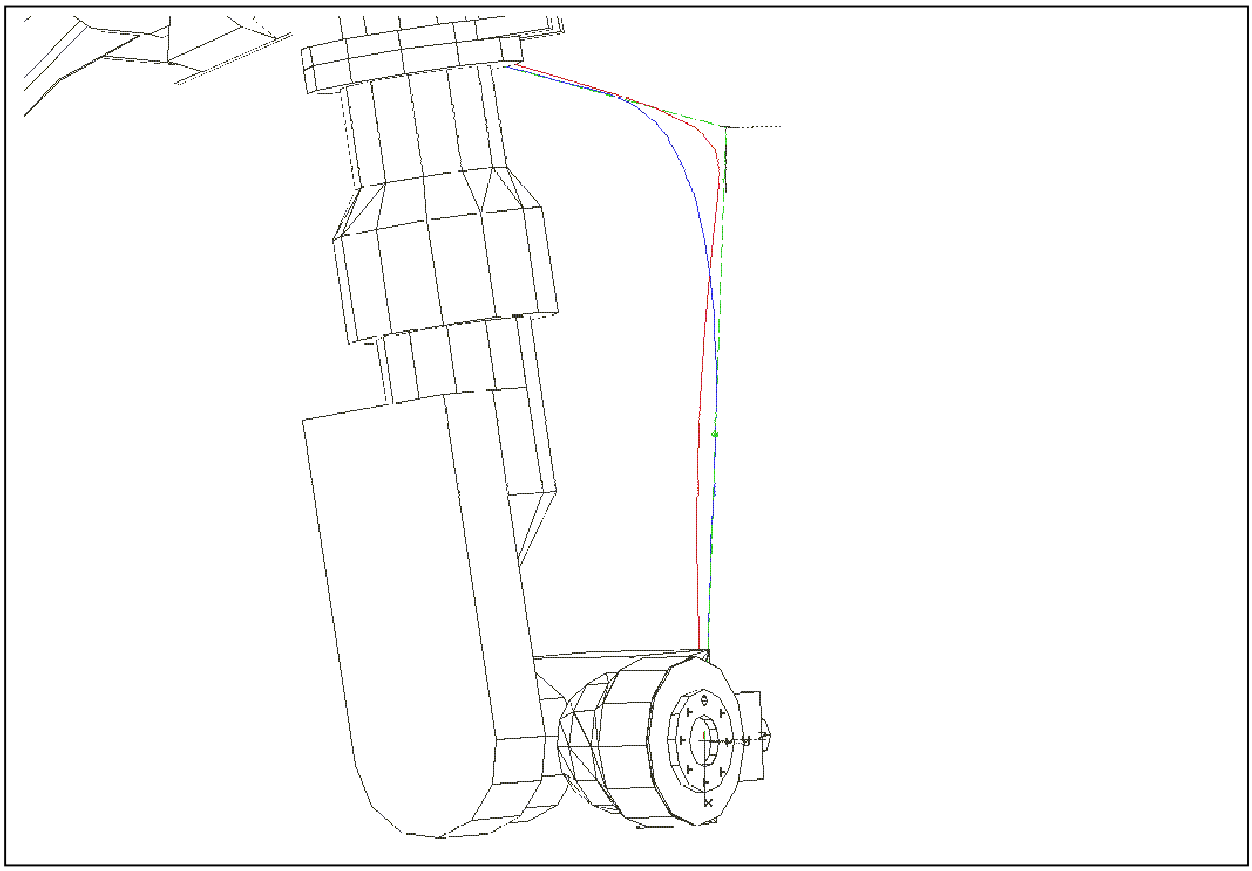


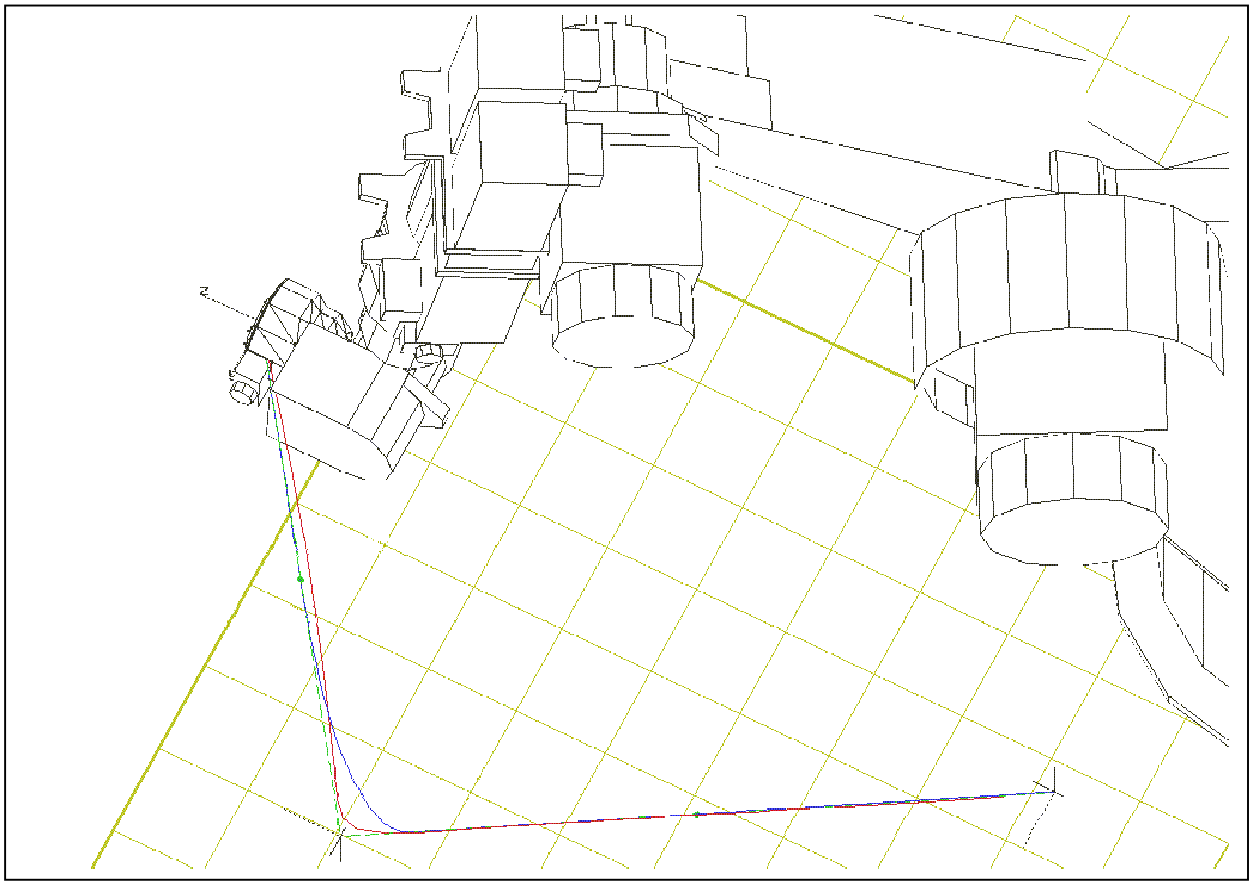





z
y
x


x
z
z
y
x





APPENDIX B : Exemplar RRS-log entries in extended format

```
1 >
Opcode:              101 (INITIALIZE )
RobotNumber:         0
RobotPathName:       "/u/dede/rrs/robots/"
ModulePathName:      "/u/dede/rrs/"
ManipulatorType:     "vk010"
CARRRSVersion:       0
Debug:               0

1 <
Status:              0 (The service is successful)
RCSHandle:           00 00 00 00 00 00 00 00
RCSRRSVersion:       101000
RCSVersion:          100000
NumberOfMessages:    0

1 >
Opcode:              104 (GET_ROBOT_STAMP)

1 <
Status:              0 (The service is successful)
Manipulator:        "VK010"
Controller:         "VRS1"
Software:           "VERSION001.9 (date : 20.05.94)"

1 >
Opcode:              116 (SET_INITIAL_POSITION)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0000000   j2: 00.0000000   j3: 00.0000000
                     j4: 00.0000000   j5: 00.0000000   j6: 00.0000000
Configuration:       ""

1 <
Status:              0 (The service is successful)
JointLimit:          00 00 00 00

1 >
Opcode:              120 (SELECT_MOTION_TYPE)
MotionType:          1 (Joint interpolation)

1 <
Status:              0 (The service is successful)

1 >
Opcode:              117 (SET_NEXT_TARGET)
TargetID:            0 (No identifier given)
TargetParam:         0 (unused, only position data are valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.3577235   j2: 00.3515908   j3: -0.7927651
                     j4: 01.1412816   j5: 01.0989452   j6: 00.0000011
Configuration:       ""
TargetParamValue:    0.000000

1 <
Status:              0 (The service is successful)
```

```
1 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

1 <
Status:              1 (Need more data)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: -0.0000000   j2: 00.0000000   j3: 00.0000000
                     j4: -0.0000000   j5: 00.0000000   j6: 00.0000000
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

2 >
Opcode:              117 (SET_NEXT_TARGET)
TargetID:            0 (No identifier given)
TargetParam:         0 (unused, only position data are valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: -0.4289043   j2: -0.4675541   j3: -0.4847407
                     j4: 00.5331523   j5: 00.4970707   j6: -2.6254949
Configuration:       ""
TargetParamValue:    0.000000

2 <
Status:              0 (The service is successful)

2 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

2 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: -0.0000000   j2: 00.0000000   j3: 00.0000000
                     j4: -0.0000000   j5: 00.0000000   j6: 00.0000000
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

3 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

3 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0000129   j2: 00.0000129   j3: -0.0000344
                     j4: 00.0000511   j5: 00.0000454   j6: 00.0000019
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
```

```
NumberOfEvents:      0
NumberOfMessages:    0

4 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

4 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0000816   j2: 00.0000816   j3: -0.0001848
                     j4: 00.0002621   j5: 00.0002525   j6: 00.0000001
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

5 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

5 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0002578   j2: 00.0002535   j3: -0.0005801
                     j4: 00.0008309   j5: 00.0008029   j6: 00.0000020
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

6 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

6 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0006273   j2: 00.0006145   j3: -0.0013922
                     j4: 00.0020070   j5: 00.0019298   j6: 00.0000071
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

7 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

7 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
```

```
JointPos axesvalues: j1: 00.0012762   j2: 00.0012547   j3: -0.0028359
                     j4: 00.0040778   j5: 00.0039307   j6: 00.0000068
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

8 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

8 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0023160   j2: 00.0022773   j3: -0.0051348
                     j4: 00.0073951   j5: 00.0071168   j6: 00.0000039
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

9 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

9 <
Status:              0 (The service is successful)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: 00.0038500   j2: 00.0037855   j3: -0.0085336
                     j4: 00.0122846   j5: 00.0118310   j6: 00.0000017
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
NumberOfMessages:    0

.
.
. get_next_step entries
.
.

108 >
Opcode:              118 (GET_NEXT_STEP)
OutputFormat:        6

108 <
Status:              2 (Final step, target reached or speed is zero)
CartPos flag:        00 00 00 00 (CartMatrix is not valid)
JointPos axesformat: 1 (angles and distances)
JointPos flags:      00 00 00 3f
JointPos axesvalues: j1: -0.4289002   j2: -0.4675505   j3: -0.4847379
                     j4: 00.5331478   j5: 00.4970652   j6: -2.6256251
Configuration:       ""
ElapsedTime:         0015.00000
JointLimit:          00 00 00 00
NumberOfEvents:      0
```

```
NumberOfMessages:     0

1 >
Opcode:               103 (TERMINATE)

1 <
Status:               0 (The service is successful)
```

APPENDIX C : List of RRS-Services sorted by operation code

101 INITIALIZE
102 RESET
103 TERMINATE
104 GET_ROBOT_STAMP
105 GET_RCS_DATA
106 MODIFY_RCS_DATA
107 SAVE_RCS_DATA
108 LOAD_RCS_DATA
109 GET_INVERSE_KINEMATIC
110 GET_FORWARD_KINEMATIC
111 MATRIX_TO_CONTROLLER_POSITION
112 CONTROLLER_POSITION_TO_MATRIX
113 GET_CELL_FRAME
114 MODIFY_CELL_FRAME
115 SELECT_WORK_FRAMES
116 SET_INITIAL_POSITION
117 SET_NEXT_TARGET
118 GET_NEXT_STEP
119 SET_INTERPOLATION_TIME
120 SELECT_MOTION_TYPE
121 SELECT_TARGET_TYPE
122 SELECT_TRAJECTORY_MODE
123 SELECT_ORIENT_INTERPOLATION_MODE
124 SELECT_DOMINANT_INTERPOLATION
127 SET_ADVANCE_MOTION
128 SET_MOTION_FILTER
129 SET_OVERRIDE_POSITION
130 REVERSE_MOTION
131 SET_JOINT_SPEEDS
133 SET_CARTESIAN_POSITION_SPEED
134 SET_CARTESIAN_ORIENTATION_SPEED
135 SET_JOINT_ACCELERATIONS
137 SET_CARTESIAN_POSITION_ACCELERATION
138 SET_CARTESIAN_ORIENTATION_ACCELERATION
139 SET_OVERRIDE_SPEED
140 SELECT_FLYBY_MODE
141 SET_FLYBY_CRITERIA_PARAMETER
142 SELECT_FLYBY_CRITERIA
143 CANCEL_FLYBY_CRITERIA
144 SELECT_POINT_ACCURACY
145 SET_POINT_ACCURACY_PARAMETER
146 SELECT_TRACKING
147 SET_CONVEYOR_POSITION

148 DEFINE_EVENT
149 CANCEL_EVENT
150 GET_EVENT
151 STOP_MOTION
152 CONTINUE_MOTION
153 CANCEL_MOTION
154 GET_MESSAGE
155 SET_OVERRIDE_ACCELERATION
156 SET_MOTION_TIME
157 SELECT_WEAVING_MODE
158 SELECT_WEAVING_GROUP
159 SET_WEAVING_GROUP_PARAMETER
160 SET_PAYLOAD_PARAMETER
161 SET_CONFIGURATION_CONTROL
162 SET_JOINT_JERK
163 GET_CURRENT_TARGETID

1000 DEBUG
1001 EXTENDED_SERVICE

APPENDIX D : ROBCAD's controller simulation modelling file and the RRS-oriented action program

Control file

```
simulation_model ${CONTROLLER}/vrs_1_sim.awk
representation_mode RPY
motion_engine rrs

robot_start
name
robot_end

path_start
path_end

wp_start

  VRS_TCP_NUM
  VRS_MOTION_TYPE_MERKER
  RRS_MOTION_TYPE
  MOUNTED_WORKPIECE_FRAME_NAME                    (ext_tool_frame)
  RRS_TOOL_FRAME:absolute                 (tool_frame)
  RRS_CARTESIAN_POSITION_SPEED
  VRS_SPEED                                       (cart_speed)
  VRS_ENDSPEED
  RRS_CARTESIAN_POSITION_ACCELERATION
  VRS_ENDACCEL
  VRS_RUCK
  VRS_FILTER
  VRS_GENAU
  VRS_ERT


  VRS_UP_CALL                 (up_call)
  VRS_ANALOG_NR                       (analog_nr)
  VRS_ANALOG_KANAL_NR                 (analog_kanal_nr)
  VRS_ANALOG_AUSGART_NR
  VRS_ANALOG_WERT
  VRS_SENSOR_NR
  VRS_SENSOR_WERT
  VRS_SUCHLAUF_FERN_NR
  VRS_SUCHLAUF_NAH_NR
  VRS_SUCHLAUF_GESCH
  VRS_PENDELN_FORM
```

```
  VRS_PENDELN_AMPL
  VRS_PENDELN_PERI
  VRS_PENDELN_WINKEL
  VRS_PENDELN_EBENE
pose
move
  OLP_STRING_NUM
  OLP_STRING_00
  OLP_STRING_01 .. _49
wp_end
```

RRS-oriented action program for simulation with ROBCAD

```
#  VRS_1 robot simulation model
#
#          Hans-Hennig Steineke   Juli, 95
#          RRS-Erweiterungen von Goeksel Dedeoglu, November 1995

BEGIN {

          ######################################################################
          # The following line should be remarked when simulating without RRS          #
          VRS_1_RRS =1
          ######################################################################

          extern_point = 0;
          __semantic_check = 1
          init_EA()
          if (VRS_1_RRS==1) {
                    init_rrs()
                    }
          }

{
# robot start
if( $1 == name){
                    }
if( $0 == $0) {
                    text = sprintf("%s ",$0)
                    command = "cat 1>" "output.txt"

                    printf("%s\n ", $0) | command
                    }
if ($1 == "simulation_start") {
          if (VRS1_RRS==0) {
                    Execute(SetEnv(MotionType(JOINT)))    # example of some default
                    Execute(SetEnv(Zone("fine")))         # setting at start
                    }

          if (VRS_1_RRS==1) {
                    print "Simulation Start" > "/dev/stderr"
                    Execute(RrsSelectMotionType(1))
                    print "   Execute(RrsSelectMotionType(1))" > "/dev/stderr"
                    rrs_last_motion_type=1
                    }
          next
          }
if ($1 == "path_start") {

          if (VRS1_RRS==0) {
                    Execute(SetAlways(RelJointAccel(1.0)))    # example of some default
                    Execute(SetAlways(RelJointDecel(1.0)))    # settings for the path
                    }
          next
          }
if ($1 == "wp_start" )      {
                                        location_name = $2 }
if ($1 == "ext_tool_frame")  {
          extern_point = 1
          print "setze basis Koorinaten =====\n" > "/dev/stderr"
          Execute(SetTcpf(CurrToolFrame()))
```

```
		next
		}
if ($1 == "tool_frame") {
		split( $2, teiler, "," )
		PI = 3.1415926
		multi = 2 * PI / 360
		R = teiler[4] * multi
		P = teiler[5] * multi
		Y = teiler[6] * multi
		if(extern_point != 1) {
			print "setze kopf Koordinaten =====\n" > "/dev/stderr"
			Execute(SetTcpf(Frame(CARTESIAN, teiler[1],teiler[2], teiler[3], RPY, R, P, Y)))
			if (VRS_1_RRS==1) {

		#Execute(RrsModifyCellFrame("TOOL",RelativeFrame(CurrToolFrame(),CurrTcpf())));
				#print "
Execute(RrsModifyCellFrame(\"TOOL\",RelativeFrame(CurrToolFrame(),CurrTcpf()))"  >  "/dev/stderr"
				}
			}
		next
		}
if ($1 == "up_call" ) {
		if ( $2 > 0) {
			unterprogramm = "upfolge"$2
			jumpflag = 1
			next
			}
		else {
			jumpflag = 0
			}
		}
if ($1 == "RRS_MOTION_TYPE") {
		if (VRS_1_RRS==0) {
			if ($2 == "1")  { motype = MotionType(JOINT) }
			if ($2 == "2")  { motype = MotionType(LINEAR) }
			if ($2 == "3")  { motype = MotionType(SLEW) }
			if ($2 == "4" &&  via_flag) { motype = MotionType(CIRCULAR-VIA)
								}
			}
		if (VRS_1_RRS==1) {
		    if (rrs_last_motion_type!=$2) {
			print "Motion Type" > "/dev/stderr"
			rrs_last_motion_type=$2
			if ($2=="1") {
				Execute(RrsSelectMotionType(1))
			print "   Execute(RrsSelectMotionType(1))" > "/dev/stderr"
			next
			}
			if ($2=="2") {
			Execute(RrsSelectMotionType(2))
			print "   Execute(RrsSelectMotionType(2))" > "/dev/stderr"
			next
			}
			if ($2=="3") {
			Execute(RrsSelectMotionType(3))
			print "   Execute(RrsSelectMotionType(3))" > "/dev/stderr"
			next
			}
			if ($2=="4" && via_flag) {
					via_flag = 1
			Execute(RrsSelectMotionType(4))
			print "   Execute(RrsSelectMotionType(4))" > "/dev/stderr"
```

```
                    next
                    }
                    print "! No motion "
                    }
                    }
          next
          }
if ($1 == "cart_speed") {
          if (VRS_1_RRS==0) {
                    Execute(SetFromNowOn(Speed($2)))
                    }
          if (VRS_1_RRS==1) {
                    # ROBCAD has only one variable (cart_speed) to indicate the speed for all
                    # motion types. In the case of PTP motion this value is a percentage, whereas
                    # with LINEAR it has mm/sec as unit. To differentiate between these two cases,
                    # an additional speed variable is needed to indicate the type of motion.

                    if (rrs_last_motion_type!=1) {
                              if (rrs_last_cart_speed!=$2) {
                                        print $1 > "/dev/stderr"
                              rrs_last_cart_speed=$2
                                        Execute(RrsSetCartPosSpeed($2))
                                        print "   Execute(RrsSetCartPosSpeed(" $2 "))" > "/dev/stderr"
                                        next
                              }
                              }
                    else {
                              if (rrs_last_joint_speed!=$2) {
                              print $1 > "/dev/stderr"
                              rrs_last_joint_speed=$2
                                        pose_val =Pose( $2, $2, $2, $2, $2, $2, $2, $2, $2, $2)
                              Execute(RrsSetAllJointsSpeed(pose_val))
                              print "   Execute(RrsSetAllJointsSpeed(" $2 "))" > "/dev/stderr"
                              next
                                        }
                              }
                    }
          next
          }
if (VRS_1_RRS==1) {
          if (($1=="RRS_CARTESIAN_POSITION_DECELERATION") || ($1=="VRS_ENDACCEL")) {
                    if (rrs_last_motion_type!=1) {
                              if (rrs_last_cart_pos_decel!=$2) {
                              print $1 > "/dev/stderr"
                              rrs_last_cart_pos_decel=$2
                                        Execute(RrsSetCartPosAcc($2,2))
                              print "   Execute(RrsSetCartPosAcc("$2",2))" > "/dev/stderr"
                              next
                              }
                              }
                    else {
                              if (rrs_last_joint_decel=$2) {
                              print $1 > "/dev/stderr"
                                        rrs_last_joint_decel=$2
                                        #pose_val =Pose( $2, $2, $2, $2, $2, $2, $2, $2, $2, $2)
                              #Execute(RrsSetAllJointsAcc(pose_val,2))
                              #print "   Execute(RrsSetAllJointsAcc(" $2 ",2))" > "/dev/stderr"
                              next
                                        }
                              }
                    }
          if ($1=="RRS_CARTESIAN_POSITION_ACCELERATION") {
```

```
                        if (rrs_last_motion_type!=1) {
                                if (rrs_last_cart_pos_accel!=$2) {
                                print $1 > "/dev/stderr"
                                rrs_last_cart_pos_accel=$2
                                        Execute(RrsSetCartPosAcc($2,1))
                                print "   Execute(RrsSetCartPosAcc("$2",1))" > "/dev/stderr"
                                next
                                }
                                }
                        else {
                                if (rrs_last_joint_accel=$2) {
                                print $1 > "/dev/stderr"
                                        rrs_last_joint_accel=$2
                                Execute(RrsSetAllJointsAcc($2,1))
                                print "   Execute(RrsSetAllJointsAcc(" $2 ",1))" > "/dev/stderr"
                                next
                                        }
                                }
                        }
                }
if ($1=="VRS_GENAU") {
        if (rrs_last_genau!=$2) {
                print "VRS Genau" > "/dev/stderr"
                rrs_last_genau=$2
                Execute(RrsSetFlybyCriteriaParam(0,$2))
                print "   Execute(RrsSetFlybyCriteriaParam(0,"$2"))" > "/dev/stderr"
                }
        }
# pose never used !!
if($1 == "pose") {
                #           printf(" pose %s ", $0) > "/dev/tty"
                command = "cat 1>" "poses.txt"
                # printf("%s\n ", $0) | "cat 1> poses.txt"
                printf("%s\n ", $0) | command
                split($2,pa,",");
                pose_val=Pose(pa[1],pa[2],pa[3],pa[4],pa[5],pa[6],0.0,0.0,0.0,0.0);
                print pose_val > "/dev/stderr"
                Execute(RrsSetNextTargetPos(pose_val))
                }

if($1 == "move") {
                if (VRS_1_RRS==0) {
                        Execute(SetOnce(motype))
                        }
                if (via_flag) {
                        if (VRS_1_RRS==0) {
                                Execute(DefineViaPoint(LocName($2)))
                                }

                        via_flag = 0

                        if (VRS_1_RRS==1) {
                        print "-Via Point" > "/dev/stderr"
                                }
                        }
                else {
                        if (VRS_1_RRS==0) {
                                Execute(DefineToPoint(LocName($2)))
                                }
                        if (VRS_1_RRS==1) {
                                #Execute(RrsSetNextTargetLoc(location_name))
                                #Execute(RrsSetNextTargetPos(LocToPose(location_name)))
```

```
                    #print " .  Execute(RrsSetNextTargetPos(LocToPose(location_name)))" > "/dev/stderr"
                        }
                    }

            if (VRS_1_RRS==0) {
                    Execute(Move())
                    }
            if (VRS_1_RRS==1) {
                    Execute(RrsExecMotion())
            print "   Execute(RrsExecMotion())" > "/dev/stderr"
                    }
            if (jumpflag) {
                    ruecksprung_loc = $2
                    Execute(CallPath(unterprogramm))

                    if (VRS_1_RRS==0) {
                            Execute(DefineToPoint(LocName(ruecksprung_loc)))
                            }

                    if (VRS_1_RRS==1) {
                            #Execute(RrsSetNextTarget(ruecksprung_loc))
                            Execute(RrsSetNextTargetPos(LocToPose(ruecksprung_loc)))
                            }
                    jumpflag = 0
                    }
            next
    }
/* -------- Example of dealing with text attributes --------- */

if($1 == "OLP_STRING_NUM") {
    olp_string_num = $2
    next
}
if( substr($1, 1, 11) == "OLP_STRING_") {
                                    line = substr($1, 12, 2)
                                    if (line < olp_string_num) {
                                deal_with_text_attr( substr($0, 15) )
                                    }
                                    }
if($1 == "pose") {  print $0 > "poses.txt" }

if($1 == "wp_end")  {
            extern_point = 0
            }
if($1 == "path_end")        { }

}  #--------- Ende des Ganzen....

function deal_with_text_attr( line_command )
{
# Dealing with the line_command can be done with
# using $0 = line_command in order to split to fields
$0 = line_command
# if ($1=="CALL") ...
# if ($1=="OUTPUT") ...

  if (substr($1, 1,1) == "A")
            {
                    if($3 == "EIN") { A_zeile[$1] = 1}
                    if($3 == "AUS") { A_zeile[$1] = 0}

                    Execute(SetSignal($1, A_zeile[$1]))
```

```
                next
                }
if ($1 == "warte")
                {
                if ($2 == "bis")      {
                        if (substr($3,1,1)  == "!") { warteauf = 0}
                        if (substr($3,1,1)  == "E") { warteauf = 1}
                        if (substr($3, 1,1) == "F") { warteauf = 1}
                        if (substr($3, 1,1) == "A") { warteauf = 1}
                        if (substr($3, 1,1) == "M") { warteauf = 1}
                        printf $2 $3 > "/dev/stderr"
                        Execute(WaitSignal($3,warteauf))
                        next
                        }
  if ($2 == "onl")
                {
                        if (substr($3, 1,1) == "!") { warteauf = 0}
                        if (substr($3, 1,1) == "E") { warteauf = 1}
                        if (substr($3, 1,1) == "F") { warteauf = 1}
                        if (substr($3, 1,1) == "A") { warteauf = 1}
                        if (substr($3, 1,1) == "M") { warteauf = 1}
printf $2 $3 > "/dev/stderr"

                Execute(WaitSignal($3,1))
                next
                }
                }
}

function init_EA()
#-------------------------------------------------
{
                moveflag = 0
                for(i=1;i<129;i++)
                        { e_name = "E"+ i
                          a_name = "A"+ i

                          E_zeile[e_name] = 0
                          A_zeile[a_name] = 0
                        }
 }
# ------------------------------------------------
function init_rrs()
{
rrs_last_joint_speed=-1
rrs_last_cart_speed=-1
rrs_last_cart_pos_accel=-1
rrs_last_cart_pos_decel=-1
rrs_last_genau=-1
rrs_last_motion_type=-1
}
```

APPENDIX E : Source code of the RCSVW-Module and its compilation

Software components of the RCSVW-Module

The RCSVW-Module is designed to run on any UNIX compatible operating system. The original development environment of the module has been an IBM RISC/6000 workstation with the operating system AIX V.3.2, whereas the target system where the CAR-Tool *ROBCAD* resides is a Silicon Graphics - Indigo workstation, operating with IRIX. The compilation and linking procedures described in this document have been successfuly performed on both systems.

The RCSVW-Module consists of five executable UNIX files, a default manipulator and an optional version data :

Executables	Data files
rcsvw10	robdaten.txt (ASCII)
inter1	version (ASCII)
bmvert	
ertv	
init1	

The program *rcsvw10* is the RCSVW-Module itself, whereas the other executables are software components of VRS1-controller's original path module.

Calling the RCSVW-Module

Since the RCSVW-Module has been designed to be integrated into a CAR-Tool according to „two module concept“, the program *rcsvw10* is expected to be spawned by the CAR-Tool. Indeed, the main() function of this program has been delivered by the company Tecnomatix Ltd. and extended for the RCSVW-Module.

The other executables making up the path module will be spawned by the module as necessary.

Compiling and linking

For each executable data (rcsvw10, inter1, bmvert, ertv and init), there is a make file, whose extension is '.mk'. After the object files are created, the *make* program will also link them altogether to produce executables.

As far as the path module is concerned, all dependencies defined in their original make files have been kept.

To compile the RCSVW-Module, one may use the batch program *make_rcsvw10*, which will simply call the standard UNIX *make* utility for all executables listed above. The compiler which will be called by *make* is *cc*, the standard UNIX C compiler.

Source files needed to compile the RCSVW-Module :

make_rcsvw10	Shell script which will call the make utility
rcsvw10.mk	Make file for the RCSVW-Module
inter1.mk	Make file for the executable 'inter1'
ertv.mk	Make file for the executable 'ertv'
init.mk	Make file for the executable 'init1'
bmvert.mk	Make file for the executable 'bmvert'
rcsvw10.c	C code of the main RCSVW-Module
rcsvw10.h	Include file for the main RCSVW-Module
rcsvw10_base.c	C code, base RRS services.
rcsvw10_moti.c	C code, motion related RRS services
rcsvw10_mopa.c	C code, RRS services on motion parameters
rcsvw10_modi.c	C code, RRS services for motion modification
rcsvw10_cond.c	C code, RRS services for condition handling
rcsvw10_weav.c	C code, RRS services for weaving
rcsvw10_base_debug.c	C code, debug functions for RRS services
rcsvw10_moti_debug.c	C code, debug functions for RRS services
rcsvw10_mopa_debug.c	C code, debug functions for RRS services
rcsvw10_modi_debug.c	C code, debug functions for RRS services
rcsvw10_cond_debug.c	C code, debug functions for RRS services
rcsvw10_weav_debug.c	C code, debug functions for RRS services
rcsvw10_strc.h	Include file for data structures
rcsvw10_defi.h	Include file for definitions
rcsvw10_ipc.c	C code, routines for interprocess communication
rcsvw10_ram.c	C code, standard routines for all services
rcsvw10_xlib.c	C code, x-library functions
rcsvw10_bm.c	C code, routines that are copied from the path module.
rcsvw10_lib.c	C code, routines common to all executables

The files listed above should all be available under the same directory, where the batch program will be called.

Original path module's files needed by the RCSVW-Module :

var-bahn.gbl
define.gbl
struct.gbl
milib.h
inter1.c
bmvert.c
init.c
ertv.c
linauf.c
linue.c
ptpauf.c
ptpue.c
splauf.c
splue.c
zirkauf.c
zirkue.c
best_tra.c
eauf.c
iniba.c
joy.c
kms.c
matlib.c
onl.c
pend.c
penini.c
pktv.c
prof.c
regler.c
sen.c
senini.c
simul.c
taskk.c
gblram.c
trans.c
sp_lokal.c