# An evaluation of the McDonnell Douglas robotics 7.0 software

Hein-Jan van Veldhoven
ID: 221176

# An evaluation of the McDonnell Douglas Robotics 7.0 software.

**Wpa nr: 1098**

Professor: Prof. Dr. Ir. A.C.H. van der Wolf

Coaches: Dr. Ir. F.J.M. Delbressine
Ing. J.J.M. Schrauwen
F.Soers

Author: Hein-Jan van Veldhoven
Heezerweg 256f
Eindhoven
Id nr: 221176

Hein-Jan van Veldhoven
ID: 221176

## Index

Hein-Jan van Veldhoven
ID: 221176

## Preface

On the 22 of june I first got into touch with the Robotics software, provide by McDonnell Douglas, Paris.
Since then a lot of water has flowed under the bridge, and I hope that by now I have finished my assignment. Though the Robotics software and the computer associated with it were pleasant to work with, I have to admit I'm glad to be able to close the book on this part of my live, because I feel that it is time for a change.
Though I am the one signing for this report, there is a number of people whom I couldn't have done without, and I feel that though it is only of small comfort to them I need to name them at this stage. These people were:

- Mr. F.L.M. Delbressine, Mr. J.J.M. Schrauwen, Mr. F.S. Soers and Mr. A.C.H. van der Wolf: For their constructive criticism.
- Henk van Rooij: In helping me convince computers of my intentions.
- John Vernooij: By explaining the program to him I got a better grasp at it.
- My parents: For supporting me in getting where I am today.
- The people from the CAD-room. For pleasant company.

The report is divided into several sections. The appendices A, B and C give an overview of the software modules BUILD, SIMULATION and COMMAND. If one is an unexperienced user of the Robotics software, it is best to first read the introduction and then read the appendices to get a good feeling for the program.
This report does not intend to replace the manuals provided with the Robotics software, but should rather be used as an supplementary to these manuals.
Though I tried to write a consistent and informative report, the reader might still have any questions concerning the software or this report. If so I will be more than willing to try and provide him/her with an answer.

Hein-Jan van Veldhoven
7th of june 1991, Eindhoven.

### 0.0.0. Summary

It was my assignment to evaluate the seventh release of the Robotics software by McDonnell Douglas with regard to its capabilities and flaws. As a test case it was decided upon a Flexible Welding Automation Cell.
The assignment meant: modelling the cell, simulating it with the software, and then perform the off-line programming with the aid of the software.
Stepping through all this I would be able to get a good feeling for the program.
Having worked with the program for several months now I feel that it can be a powerful aid in performing off-line programming for robots.
The major advantage of the program is its ability to simulate in a correct way a number of largely differing situations. This may range from a welding cell to a transfer-line, but also from a machining-centre to a human ergonomics simulation.
The only major drawback to the program is its unability to simulate devices with less then six degrees of freedom in an easy and correct manner.
All in all, however, I feel that this does not weight-up to the advantages the program can provide.

Hein-Jan van Veldhoven
ID: 221176

## 1.0.0. Assignment

Eindhoven University of Technology
Den Dolech 2
5612 AZ Eindhoven, The Netherlands
Faculty of Mechanical Engineering
Production Engineering and Automation

Eindhoven, 30 october 1990.

| | |
|---|---|
| Thesis | H.J.M. van Veldhoven |
| Chair | Prof. Dr. Ir A.C.H. van der Wolf |
| Coaches | Dr. Ir F.L.M. Delbressine<br>Ing. J.J.M. Schrauwen<br>F.S. Soers |
| Subject | Robotics 7.0 software |

Background:

In a modern day production environment, with short series and large ranges of products, robots are a necessity in trying to produce these products in a competitive way. However, it takes a lot of time to reprogram these robots if an other range of products is being made. Reprogramming these robots off-line was rather difficult until now, because the off-line programming system was not graphically based. The Robotics software from McDonnell Douglas is a graphically based off-line programming and simulation system.

Assignment:

Examine and evaluate the Robotics 7.0 software from McDonnell Douglas.

The examination will be done in a number of phases:

- Getting to know the program.
- Simulating a welding cell currently being present at the University, and consisting of:
  - A Kuka Robot with welding equipment
  - A Kuka manipulator.
- Perform off-line programming of the robot and the manipulator, including the writing and or testing of the post-processor for the welding cell.

During this assignment it is understood that McDonnell Douglas will be notified of any bugs found in the Robotics 7.0 software.

1.1.0. Comments

Reviewing the work of the past months I have been working on this assignment, it has become clear to me that though the cell I have been simulating may not have been representative for the cells which normally would be simulated with the aid of this software, it nevertheless provided the same problems which might occur in trying to simulate a cell which might be more representative.
The cell may be seen as a typical example. Their may been two sorts of industrial companies interested in this software, one sort of company which has a lot of robots that all need reprogramming in a relative short amount of time e.g. car-manufacturers, and one sort of company which may not have all that much robots, but uses these robots in a flexible way and in complex cell layouts e.g. small flexible automated cells.
The cell I used to test this software on, clearly falls into the last category.

## 2.0.0. Introduction

Robotics

The Robotics software is a set of separate programs provided by McDonnell Douglas, for the development of a computer based simulation- and off-line programming environment. To provide these functions the Robotics software consists of a number of separate programs, communicating with each other via a small number of files. These programs are:

1. BUILD
2. PLACE
   - SIMULATION
3. COMMAND
4. ADJUST
5. Cycle Time Analyzer (CTA)

Though these programs make up the complete Robotics package, it is not necessary to use all of them in a typical Robotics session.

The Robotics software can be used for more than one purpose, but it was originally designed as a computer based graphically off-line programming system for robots. The system is relatively unique in its kind, as most off-line programming systems are text-based. The system was developed independent of any type or make of robot, and in that way the system can provide its services independent of any type or make of robot. This may be especially useful if more than one type or make of robot has to be simulated and programmed.

In a production environment where more and more CAD-systems are being used to integrate the design and manufacturing process, it may be worth while to check upon a system which can integrate the programming of the robots into this process. Of course such a system should be able to address any data already created on the CAD/CAM systems.

As this was our first experience with the Robotics software, I have concentrated myself on the three most important programs, to me, namely: BUILD, SIMULATION, and COMMAND. Underneath you will find an introduction into these programs, a more extensive description of these programs can be found in appendices A, B, and C. The way the programs cooperate with each other can be seen in figure 1, the robotics system overview.

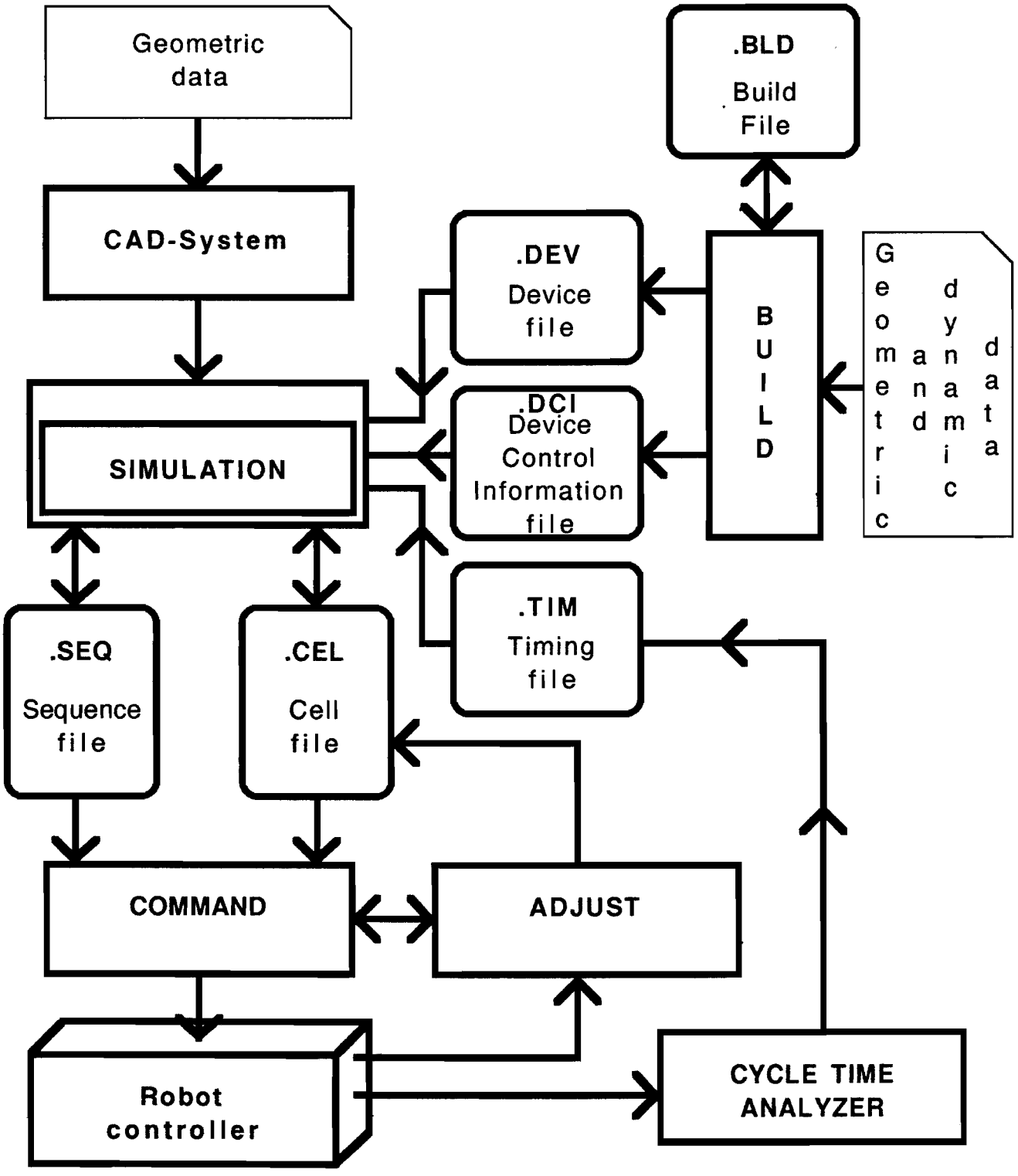


Figure 1: Robotics system overview.

The actual cell I used as a test case for the Robotics software is found in the photo underneath:

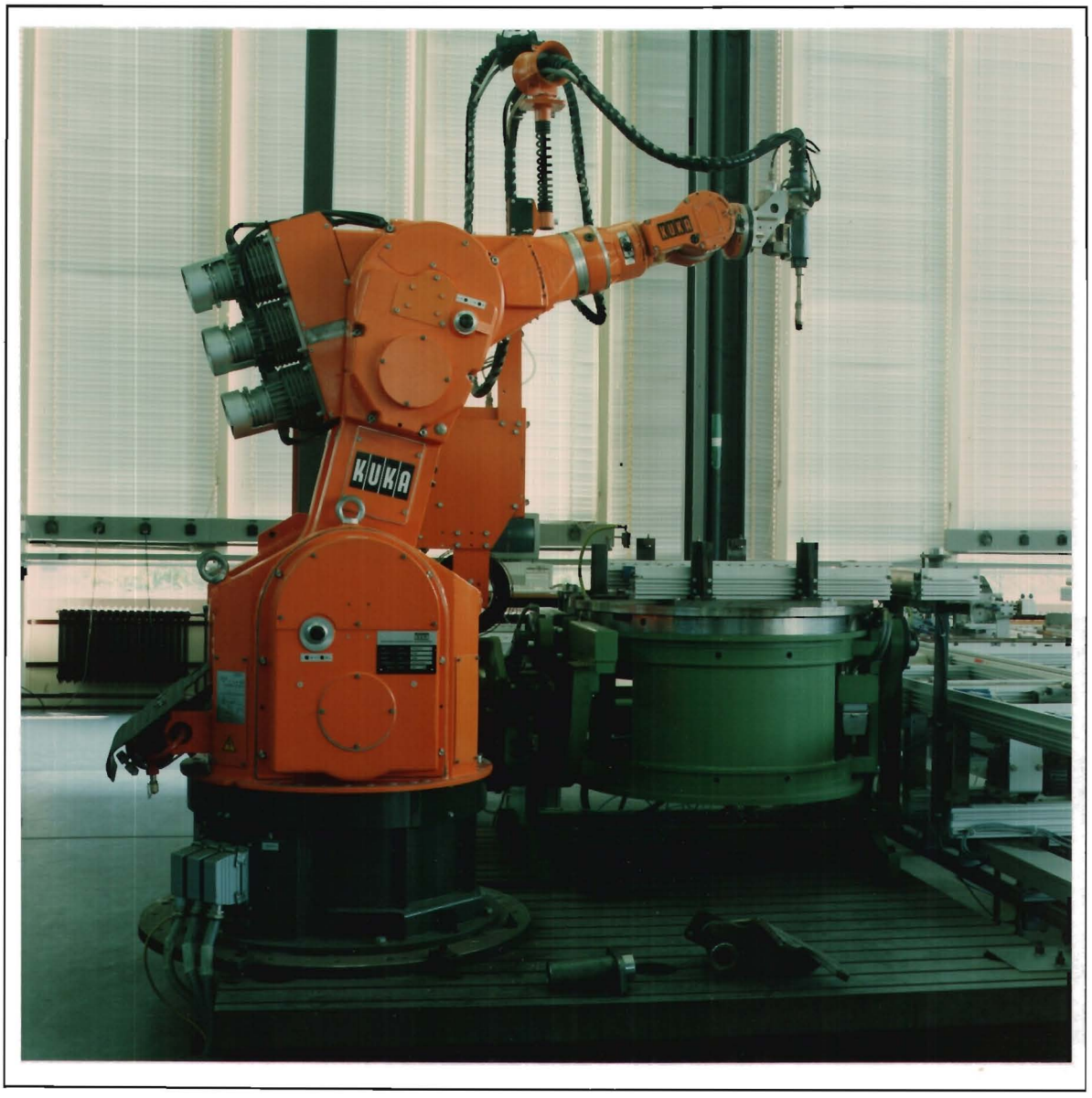


Photo 1: The actual cell

The model created of this actual cell can be found in the photo underneath, noted should be that I left out most of the welding equipment apart from the torch, and that I left out the transportation system related to the cell. This was done because I felt that it would not be important, and would cause more inconvenience than benefit.

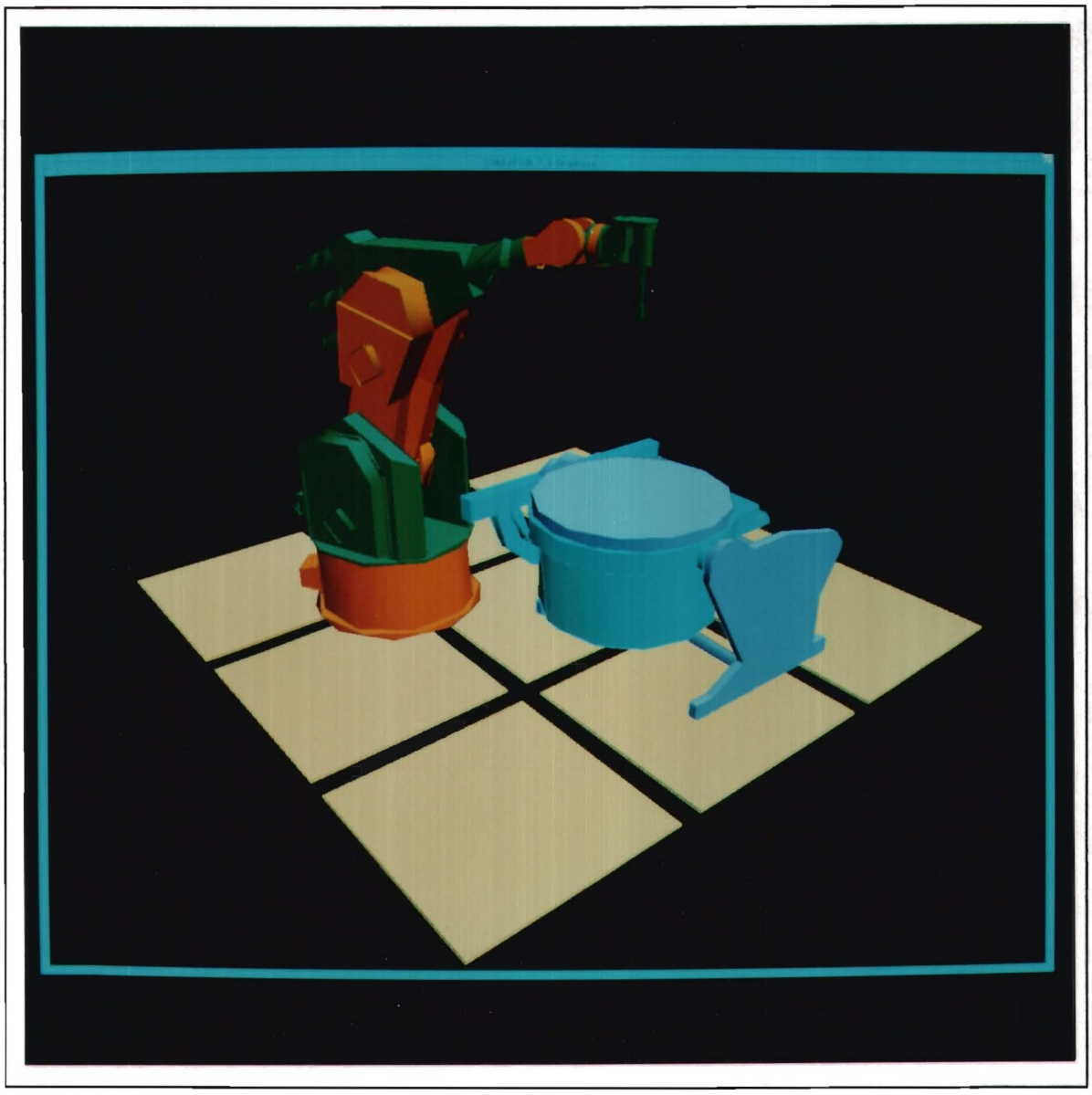

Photo 2: The model of the welding cell.

## 2.1.0. BUILD

The BUILD Robotic Applications Software Module [1] is one of the Robotics products sold by McDonnell Douglas. BUILD enables the user to quickly describe new robots or other kinematic motion devices in high-level terms. Using BUILD, the geometric model of a robot or device is automatically combined with its unique kinematic description for animation in SIMULATION. With only a minimal knowledge of robot kinematics, You will be able to add new robots or devices to the SIMULATION library. The data output files from BUILD eliminate the need to perform custom kinematic analysis.
BUILD is capable of automatically generating the forward- and inverse kinematic equations for devices which have up to 6 degrees of freedom.
Forward kinematics is an algorithm that, given a set of joint angles, computes the position and orientation of the tool and/or the faceplate of the robot.
Inverse kinematics is an algorithm that, given a position and orientation of a tool or faceplate, computes the matching joint angles of the robot.
BUILD supports devices with up to 12 degrees of freedom, but it does not automatically generate the inverse kinematic control equations. The BUILD user may optionally write a program which defines the kinematic equations for any device with up to 12 degrees of freedom. To obtain complete functionality, the device must be separable into arm and wrist components. This functionality is discussed in more detail later in section A.2.0.0. "Device Types Suitable for BUILD".
The software for the BUILD module uses the same type of menus as SIMULATION. The user must fill in each menu item in order to fully describe the characteristics of the device. Once all of the information has been entered, you may direct BUILD to create the files which contain the description of the device. Then you will be able to use the new robot or device in SIMULATION.
The steps required for creating a robot or device are as follows:

1. The user creates the geometric description or model or a robot or device on a CAD system. (Preferably UG-II) ←
2. The user collects the input data needed for the BUILD module.
3. The user transfers the CAD data to the SIMULATION data-base.
4. The user inputs the parameters which define the device, into BUILD.
5. The BUILD software generates the kinematic data which is used to simulate the motion of the robot or device in SIMULATION.
6. The users tests the robot in the SIMULATION model.

BUILD directs the user to define the kinematic model of a device, which is used by SIMULATION to drive the simulation. Additional controller information is supplied which is used by COMMAND to provide for off-line programming. BUILD will allow the user to define robot type devices as well as probe devices. A Probe is a special type of device which may be used with the ADJUST module for cell calibration.
BUILD places restrictions on the types of linkages which may be automatically modeled with complete forward and inverse kinematic solutions. If a device does not meet these restrictions, it may still be modeled, but in a much more limited way, BUILD will not be able to support the inverse kinematic solutions and therefore only simple joint motions will be supported. The user may retain complete functionality by writing a program which supplies the inverse kinematic algorithms needed for simulation.

## 2.2.0. SIMULATION

SIMULATION [2] is a software package designed to create, analyze, and modify robotic cells graphically through the use of a high speed, vector refresh or raster colour graphics display station. SIMULATION allows one to determine if required motions can be accomplished by the robot and if estimated cycle times can be met.
To accomplish this, SIMULATION offers:

1. Kinematic equations that simulate the motion of individual robots
2. Continuous readout of joint angle data that tracks the path of the robot's motion
3. Hierarchical definition of connected parts
4. Computer aided design data that geometrically describes cell components
5. Powerful 3-D colour graphics for manipulation and placement of cell components
6. Hardware controlled dynamic 3-D scaling, translating, and rotating of a view
7. Definition of motion sequences for cell analysis and collision detection
8. An expanding library of more than 130 of the most common robots
9. An optional cycle time package that provides more accurate timing of robot moves in SIMULATION by compensating for acceleration and deceleration
10. Simulation of two or more independent robots or objects moving at the same time on the screen (parallelism)
11. Explicit synchronization of the motion of two or more devices (compound device)
12. Metric support
13. Simulation of sending, receiving, and testing signals by a robot controller
14. Insertion of robot-dependent commands directly into motion sequences for simplifying off-line programming
15. Automatic collision detection.

SIMULATION uses wire frame and facet-faced graphic display models to represent robots, equipment, workpieces, and tooling within a manufacturing cell. The user can easily position these graphic models within the cell either individually, as in the case of a workpiece, or together, as in the case of a multi-segmented robot.
SIMULATION utilizes the concept of a generalized tool tip called the WORKING TPOINT. You command a robot to a desired location in terms of this WORKING TPOINT. The system determines robot joint angles so that you need not be concerned with the low-level details of robot motion. In addition, SIMULATION has a tracking option that allows you to analyze the path of the robot's motion for specific applications such as a conveyor line.
The SIMULATION software displays equipment positioning and robot motion trajectories smoothly at the robotics design station. SIMULATION continually monitors joint limits for the particular robot performing motion. In addition to joint errors, proximity of the robot to a joint limit is also recorded.
Visual collision detection is available with SIMULATION. You can change views of the cell smoothly from any angle or use the zoom feature to get a closer look. With these powerful viewing capabilities, collisions between various items within the cell can be identified quickly and avoided.
Automatic collision detection is also available. SIMULATION computes the convex hulls of the parts. Interference checking during any kind of motion is done on those convex hulls.
After a cell has been laid out to your satisfaction, various design data can be requested from SIMULATION. The complete spatial relationship between any two points, two frames, or between a point and a frame is readily accessible to the user.

To accommodate a varied set of robotic applications, SIMULATION maintains a library of commercially available robots. You can request any robot in the robot library for evaluation and motion analysis on the robotics terminal. If you have a robot that is not in the library or have a completely new robot ( or "n" degree of freedom manipulator), then that new robot must be inserted into the SIMULATION library before it can be used. This can be accomplished using the BUILD program and a CAD-package, for instance Unigraphics II.

2.3.0. COMMAND

COMMAND [3] is a software module used for programming robots off-line. It is used in conjunction with SIMULATION and a specific robot translator to generate a complete robot program that can be loaded and executed on a particular robot's control unit.
To accomplish this, COMMAND:

1. Utilizes sequence-files created with SIMULATION for robot motion,
2. Utilizes sequence-files created with SIMULATION for robot I/O,
3. Provides full use of the robot's functions in its native language,
4. Provides full program annotation.

The user directs COMMAND to merge a SIMULATION sequence with a complete robot program via a file known as the 'User Program File' or User File. The user file is created with a standard text editor. It may include robot language instructions which are specific to the target robot, and it may include Sequence Access Commands. Sequence access commands are used to direct COMMAND to include SIMULATION sequences into the robot program.
In order to generate a meaningful and complete robot program, the user must have access to all features, functions and options that a particular robot controller supports. The User File gives you this capability by allowing you to work in the native language of your robot. Motion and I/O can then be extracted from SIMULATION sequences and added to the program. Optionally you may include robot specific instructions in the SIMULATION sequences, SIMULATION will ignore these instructions, while COMMAND will include them in the robot program.
A powerful feature of COMMAND permits the user to associate groups of logic instructions with specific robot positions generated by SIMULATION sequences. For example, the user may define a subroutine in the robot's native language, to open and close a gripper. This subroutine is identified by an operation name, and easily referenced in SIMULATION sequences by tagging the desired robot motion commands with that operation name. This will result in the gripper being opened or closed at the correct location in the program.

## 3.0.0. Creating devices in Robotics

The first item one encounters when one starts simulating cells in Robotics is having to create the simulation environment. If one is lucky it is possible to merge one of the robots from the robot system library supplied with the Robotics software. If, however, one has to create one's own robot or manipulator the following steps have to be taken:

1. The gathering of data on the device, such as joint-types, joint speeds, distance between joints etc.
2. Creating a model of the device.
3. Converting the device model to Robotics.
4. Creating a device description using BUILD.
5. Checking the device.

### 3.0.1. Gathering device data:

The required device data can usually be obtained from the device instruction manuals, or from the device manufacturer. Most of the data will be in the right format for incorporation in the device description, some of the data, however, may have to recomputed before it can be incorporated in the device description.

### 3.0.2. Device model creation:

In order to be able to simulate a device in a proper manner, a device model should be accessible to the Robotics software. The device model is made accessible to the Robotics software through a conversion utility.
The device model has to be created before it can be converted. The creation of the device model is done preferably in UNIGRAPHICS II. This is because though there is an IGES to SIMULATION conversion utility, documentation shows it to be a more limited conversion utility than the UNIGRAPHICS II to SIMULATION utility. However, I have not been able to test the IGES to SIMULATION utility.
The device is modeled, link by link, with each consecutive link modeled in such a way that the axis of motion coincides with one of the axis of the absolute coordinate system. The easiest way of creating this effect is by modelling the complete device with each separate link on a separate layer, and later translating these links to the absolute coordinate system. BUILD expects the device to be created in this manner, so the transformation between the consecutive links can be done by BUILD and can be incorporated into the forward- and inverse kinematic algorithms.
When modelling the device the user should be cautious in using wire-frame elements, the conversion routine will not convert wire-frame elements into solid shaded images, there is however, a possibility to convert certain surface into solid shaded images.

### 3.0.3. Converting the device model:

After the device has been completely modeled, it has to be converted. The conversion may be done with, for instance, the UNIGRAPHICS II to SIMULATION utility. The user should be aware that wire frame drawings cannot be converted into solid shaded images, if the user wants solid shaded images, the device should be modeled with solids or certain types of surfaces, instead of wire frames.

A second pitfall in converting devices is the facet tolerance. The user may easily be tempted to create parts with too many facets (a small facet tolerance). If the user tries to merge such a part or device into a cell, the program may bomb out and create an out of memory error. The easiest solution to this is reconverting the parts with a larger facet tolerance. (Less facets).
The technique used to create the solid shaded image is as follows:
The surface of the part is divided into facets, a light source is placed in a certain position and direction. The normal vector of the facet and the normal vector of the light source are computed. The two vectors are compared and a value for shading is derived.
The drawback of this method is that it doesn't take any other parts into account which might obstruct or reflect the light beam. The advantage of course is that it is relatively quick in computing the images.

3.0.4. Creating a device description using BUILD:

The device description for SIMULATION is created with the aid of BUILD. Here all the data collected in 3.0.1. is structured and used to create a unique device description.
All the kinematic transformations, arm configurations, allowable motion modes, tool tpoints, off-line programming coordinate systems and sources of inverse kinematics are generated or selected in order to create an as accurate as possible model of the robot.

3.0.5. Testing the device:

The fist step in testing the device is merging it into the cell, the user will than be able to see whether all the constant transformation were correctly inputted.
After it has passed this initial test, the user may invoke the Goto Joint Position function to check whether all the joints move in the correct manner. This will mean the forward kinematics is correctly defined.
Testing the inverse kinematics is a lot more difficult: An indication can be obtained by creating an arbitrary tpoint, and have the robot align with this arbitrary tpoint, by means of a Goto Tpoint command. This will indicate whether the inverse kinematics is correct or not. It will, however, not provide a 100% guarantee of complete functionality of the device. Further testing will not improve this functionality, so after these initial test the device is ready for incorporation into the desired cell environment.

3.1.0. Device with less than six degrees of freedom

If a device with less than six degrees of freedom has to be modelled, a problem might occur in trying to simulate this device in Robotics.
An example of a device with less than six degrees of freedom is a manipulator for welding-operations.
The problem with the inverse kinematic algorithms used by SIMULATION is that they expect the first three joints of a device to be for reaching a certain position, and the last three joints of a device for obtaining a certain orientation.
This means that the distance between the first three joints of a device may not be zero if they are rotational joints.

In the case of the welding manipulator, the manipulator and the mounted parts assume a certain orientation, and in that way simplify the control of the welding process. This is not a position but an orientation, but as the manipulator only has two joints, the inverse kinematic algorithms will use these joints to reach a position. The program will bomb out leaving a rather useless error.
The way to solve this problem is to create a device with six joints, for instance three translational joints, and three rotational joints which has great similarity with the two joint device. The number of links should be the same as well as the fixed and variable translations. What one really does is insert a number of joints into the device description until the device contains six independent joints (six degrees of freedom).
The next step is to modify the two-joint-device description so that it gets its inverse kinematics from the six-joints-device by ways of a so called 'Joint Mapping Coordinate System File'. This file selects which joint from the six-joint-device should be mapped onto the two-joint-device and vice-versa.
Now SIMULATION will compute the correct joint angles for the six-joint-device, and use these on the two-joint-device. The user should be careful that translational joints can only be used as the first three joints. See also section A.2.2.0. Device Requirements for more information.

## 4.0.0. Forward- and Inverse Kinematic Algorithms

The technique used in computing the Forward- and Inverse Kinematic Algorithms in SIMULATION is a technique known as homogeneous transformations. The technique is based on vector and matrix mathematics. For more information on this subject see [4].

4.0.1. A vector:

A vector can be represented as a column matrix.

A point vector

$$\mathbf{v}=a\mathbf{i}+b\mathbf{j}+c\mathbf{k}$$

Where **i,j,** and **k**, are unit vectors along the x, y, and z coordinate axes, respectively is represented in homogeneous coordinates as a column matrix

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

where

$$a=x/w$$
$$b=y/w$$
$$c=z/w$$

4.0.2. The dot- and cross product:

Two products are defined: the dot- and cross products.
Given two vectors:

$$\mathbf{a}=a_x\mathbf{i}+a_y\mathbf{j}+a_z\mathbf{k}$$
$$\mathbf{b}=b_x\mathbf{i}+b_y\mathbf{j}+b_z\mathbf{k}$$

The dot product is indicated by a dot '.' and defined as follows:

$$\mathbf{a}.\mathbf{b}=a_xb_x+a_yb_y+a_zb_z$$

The dot product of two vectors is a scalar. The cross product, indicate by a 'x' is another vector perpendicular to the plane formed by the vectors of the product and is defined by:

$$\mathbf{a} \times \mathbf{b}=(a_yb_z-a_zb_y)\mathbf{i}+(a_zb_x-a_xb_z)\mathbf{j}+(a_xb_y-a_yb_x)\mathbf{k}$$

4.0.3. A plane:

A plane is represented as a row matrix:

$$\mathcal{P}=[a,b,c,d]$$

such that if a point **v** lies in a plane $\mathcal{P}$ the matrix product

$$\mathcal{P}\mathbf{v}=0$$

or in a expanded form

$$[a,b,c,d]\begin{bmatrix}x\\y\\z\\w\end{bmatrix}=xa+yb+zc+wd+=0$$

4.1.0. Transformations

A transformation of the space **H** is a 4*4 matrix and can represent translation, rotation, stretching, and perspective transformations. Given a point **u**, its transformation **v** is represented by the matrix product:

$$\mathbf{v}=\mathbf{H}\mathbf{u}$$

4.1.1. General Transformations:

In general a transformation **T** will look something like:

$$T=\left[\begin{array}{ccc:c} & X & & Y \\ \hdashline 0 & 0 & 0 & 1 \end{array}\right]$$

with X being the rotational part of the translation:

$$X=\begin{bmatrix}n_x & o_x & p_x\\ n_y & o_y & p_y\\ n_z & o_z & p_z\end{bmatrix}$$

and Y being the translational part of the transformation:

$$Y=\begin{bmatrix}a_x\\a_y\\a_z\end{bmatrix}$$

the bottom row of the transformation can be used for transformations such as perspective, but this is not used in robot kinematics.

4.1.2. Translation Transformations:

The transformation **H** corresponding to a translation by a vector ai+bj+ck is

$$\mathbf{H}=\text{Trans}(a,b,c)=\begin{bmatrix}1&0&0&a\\0&1&0&b\\0&0&1&c\\0&0&0&1\end{bmatrix}$$

4.1.3. Rotation Transformations:

The transformation corresponding to rotations about the x, y, or z axes by an angle Ω are:

$$\text{Rot}(x,\Omega)=\begin{bmatrix}1&0&0&0\\0&\cos\Omega&-\sin\Omega&0\\0&\sin\Omega&\cos\Omega&0\\0&0&0&1\end{bmatrix}$$

$$\text{Rot}(x,\Omega)=\begin{bmatrix}1&0&0&0\\0&\cos\Omega&-\sin\Omega&0\\0&\sin\Omega&\cos\Omega&0\\0&0&0&1\end{bmatrix}$$

$$\text{Rot}(x,\Omega)=\begin{bmatrix}1&0&0&0\\0&\cos\Omega&-\sin\Omega&0\\0&\sin\Omega&\cos\Omega&0\\0&0&0&1\end{bmatrix}$$

4.2.0. Coordinate Frames

We can interpret the elements of the homogeneous transformation as four vectors describing a second coordinate frame. The transformation matrix describes the three axis directions and the position of the origin of a coordinate frame rotated and translated away from the reference coordinate frame.

4.2.1. Relative Transformations:

The rotations and translations I have been describing have all been made with respect to the fixed reference coordinate frame. If we post-multiply a transformation representing a frame by a second transformation describing a rotation and/or translation, we make that translation and or rotation with respect to the frame axes described by the first transformation. If, however, we pre-multiply the frame transformation by a transformation representing a rotation and/or translation, then that translation and/or rotation is made with respect to the base reference coordinate frame.

4.3.0. Inverse Transformations

Now the inverse transformation can be defined as the transformation which carries the transformed coordinate frame back to the original coordinate frame. It is simply the description of the reference coordinate frame with respect to the transformed frame.
In general, given a translation with the elements:

$$T = \begin{bmatrix} n_x & o_x & p_x & a_x \\ n_y & o_y & p_y & a_y \\ n_z & o_z & p_z & a_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then the inverse transformation is:

$$T^{-1} = \begin{bmatrix} n_x & n_y & n_z & -\mathbf{a}.\mathbf{n} \\ o_x & o_y & o_z & -\mathbf{a}.\mathbf{o} \\ p_x & p_y & p_z & -\mathbf{a}.\mathbf{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where **n**,**o**,**p**, and **a** are the four column vectors and "." represents the dot product.

As the 4th row of the transformation in rotation and/or translation transformations is always 0 0 0 1 it is left out.

SIMULATION knows two transformation 3*4 matrices, being:

1. **TptoBs**. Tool with respect to Base matrix.
2. **FptoTp**. Faceplate with respect to Tool matrix

The first matrix, the **TptoBs**, is computed by post-multiplying each successive translation and/or rotation. If, for instance, a robot has six links and each transformation from link n-1 to link n can be described by a matrix $\mathbf{A_n}$ then the **TptoBs** can be defined as:

$$\mathbf{TptoBs} = \mathbf{A_1 A_2 A_3 A_4 A_5 A_6 (FptoTp^{-1})}$$

The second matrix, **FptoTp**, is defined as the faceplate with respect to the tool.

I think this separation has been made to simplify computations. The users wants the robot to align to a certain tpoint with its tool, and therefor the **TptoBs** transformation needs to be evaluated. To compute the joint angles of the robot the position and orientation of the faceplate needs to be know, and can be compute with the aid of the **FptoTp** transformation.

The matrices we have used until now can be used in forward kinematics, this means, given a set of joint angle computing the position of the tool and/or the faceplate of the robot.

Forward kinematics is the easy part, but inverse kinematics, computing the joint angles of the robot given a position and orientation, is a bit more difficult. It is not my intention to go into this to deep as I don't know which algorithms exactly are used by SIMULATION.

Obtaining a solution for the joint coordinates requires intuition and is one of the most difficult problems. The joint coordinate solutions can be obtained by equating transformation expressions. For each transformation expression we obtain 12 non-trivial equations and it is these equations which will yield the required solution. The solutions is obtained in a sequential manner, isolating each variable by pre-multiplication by a number of the transformations in each equation.
The solution is basically trigonometric in its nature. The method makes use of homogeneous transformations which provide equations for all rectangular components, both sin and cosine of all angles. These component equations are then combined with the exclusive use of the arc tangent function in order to avoid problems of angle quadrant ambiguity inherent in trigonometry.
The whole solution process is further complicated with the configuration solution. A robot may be able reach a certain position and orientation with a limited number of arm configurations. The user however wishes the robot to reach a certain position and orientation with only one arm configuration valid. This of course puts a further restriction on the way the joint angle are computed.
The user should be extremely aware of the fact that the kinematic solutions calculated by SIMULATION may not coincide with the way the robot controller solves these kinematic equations. Caution has to be taken in assuming that the algorithms, used by SIMULATION and the robot controller, always yield similar results.

## 5.0.0. Off-line Programming

Introduction.

In performing off-line programming the first thing that is needed is an accurate cell description. This includes a accurate device description and correct placements of parts.
The procedure for creating a robot specific program is fairly simple:

1. Using SIMULATION create the tpoints (positions and orientations) later needed in the program to perform the correct robot motions.
2. Using SIMULATION create a sequence describing the process that needs to be programmed.
3. Start the COMMAND module, and perform the translation from sequence to robot specific languages.

After all this has been a final check of the robot program is advisable, and it can then be loaded onto the robots controller.

### 5.1.0. Creating Tpoints

The tpoints that are needed to create the off-line programming need to be defined first. This is done in the SIMULATION module. There are several ways of creating the tpoints, for instance by creating it at the World Coordinate Frame and then translating and rotating it to a desired position, but I prefer a method in which joint values are dictated to the robot, and once it has reached a desirable position, a Tpoint is created in that position. This last method will prevent one from creating tpoints a robot might not be able to reach.

### 5.2.0. Creating sequences

After the tpoints have been created SIMULATION can be used to create a sequence. A sequence is a series of SIMULATION commands needed to perform a complex operation. As this sequence will later be used to create the robot program the user should take care that it performs the exact motions required for a certain operation.

### 5.3.0. Converting sequences to robot programs

In converting sequences to robot programs a small problems occurs at the installation present at Eindhoven, University of Technology. Though we are currently working will the 7.0 release off the Robotics software, running on a UNIX based system, we are not yet in the possession of a translator for SRCL for a UNIX based system.
In trying to perform off-line programming for the KUKA robot and its manipulator we need a translator for SRCL, luckily it was present on the VAX/DCL system.
The route I had to take was as follows:

- First I had to create the tpoints and sequences as described above.
- Then I had to start the COMMAND module on the HP (UNIX system). Using the COMMAND module I had to create a so called .CSP file (Command Source Program file). This file contains all positions and orientations and all instruction to carry the operation. It is, however, still in a robot independent language.
- Having created this .CSP file I had to transfer it to the VAX/DCL system, in order to be able to perform the next part of the conversion.

- After having transferred the .CSP file to the VAX/DCL system I started COMMAND on the VAX/DCL system, and perform the conversion form .CSP-file to robot specific file .SRC

The procedure is in fact relatively simple and I have to admit it was easier then I expected it to be.
For readers with interest in the mapping of sequence commands onto robot specific commands (SRCL) I included a superficial mapping and a list which explaining the SRCL commands, in appendices D and E.

A photo of the cell during execution of a sequence file can be found underneath:

Photo 3: The cell in operation.

## 6.0.0. Conclusions

After having worked with the Robotics software for something like several months, I have grown accustomed to it being available to solve certain problems, but also to it being around to cause them.
I tested the software in its 7.0 th release on a HP 9832 SRX computer, any other specifications of the software or of the computer may lead to different results.
I feel it would not be productive, at this stage, to given a summary of the mistakes I found, or made. I feel that it would be more productive to report on the advantages, and drawbacks of the Robotics software, so I will do just that underneath.
It should be stressed that as this program is relatively unique in its kind, this is not a comparison of this software with any other software, but solely an evaluation of the software.

### 6.1.0. Advantages

- As the program is graphically based, it is very clear exactly what the robot is being programmed to perform, so a more unambiguous robot off -line program can be created.
- As the robot programmer only has to concern himself with creating the sequences needed to program the robot, he doesn't have to worry about what language is needed to program the robot-controller. [The Robotics software is controller-language independent.]
- As most of the programming can be done off-line the robot can be productive, while new programming is being created.
- As a whole cell can be taken into account when programming the robot, most problems, such as trying to reach an impossible position or orientation, can be prevented.
- Collision detection can be used to prevent them from really happening.
- The excellent possibilities of simulating the parallel execution of programs and the signals between the devices, of the software allow one to fully model and program even such a complex thing as a transfer-line.
- The program has a menu driven command structure, so there is no need for long command strings to be entered by the user.
- As it is possible to create joints which may be dependent on or restricted by other joints, virtually any device can be modeled.
- The use of the software is not limited to robots, but it can be used for modelling mechanism, performing ergonomics simulations (accessability and/or human motion) and assembly simulations (can a certain part be manoeuvred into position). Or, for instance simulating a five axis machining-centre as is currently being done at the University.
- When creating the off-line programming, errors and warnings are flagged.
- The software is robot manufacturer independent, so a large number of robots can be simulated and off-line programmed with the systems as long as a translator for the robot can be obtained.

6.2.0. Drawbacks

- The software seems to have a number of problems in trying to create a device with less than six degrees of freedom. Though no problems are signalled by BUILD, SIMULATION bombs out in trying to move some of these types of devices. The error it leaves gives no clue as to what action can be taken to prevent these error from happening.
- The BUILD module seems to be polluting its own files. After have made a number of changes in a device description, the files still contain "old" data which has not been erased properly, this data cause problems in the SIMULATION module.
- In some parts of the program (for instance Colour Frame/Tpoints) the Terminate Operation button restores the original situation, but in other parts of the program (such as Create Tpoint), it exits the function but doesn't restore the original situation.
- The graphics seem to fill up the computers memory rather fast. Trying to perform high-definition graphics of a number of frames is not possible. The user should be aware of these problems.
- The UNIX structure in combination with the X11 window system, do not allow the user to input data or use function buttons, if the pointer is not located in the Menu Window. This can be a nuisance sometimes.
- The computer capacity needed to run the software is big. The system requires a large investment.

6.3.0. Comments on the assignment

I feel that the system could best be used by a company which has a lot of robots that all simultaneously need reprogramming in a short amount of time. This is for instance the case with transfer lines for welding car bodies. The system may also prove beneficial in a situation where a relatively small number of robots in a complex environment need a lot of reprogramming.
The situation I used to test this program, a welding cell which should be flexible, is rather different from the first situation I have just indicated, but is relatively similar to the second situation.
It may be clear that using this program for only one robot installation could be considered as overkill, by this I mean that the systems is far to powerful to be dedicated to only installation. However, I feel that most of the problems I ran into are not all that different from the problems anyone would run into trying to simulate any other environment, including transfer lines. I feel that in trying to asses the quality of the program the situation I used would do as good as any other.

6.4.0. General conclusions

The program did live up to the expectations it raised with me. It can definitely be a useful aid in off-line programming of robots and other simulation exercises. Though I feel its main attraction lies in the off-line programming, once a software package like this has been acquired by a company other might be run on the system.
The most important drawback to the system I found its unability to easily simulate device with less than six degrees of freedom (such as manipulators).
I was in the pleasant position of being able to test the program without being bothered by the cost of it. In my analyses of the program I did not incorporate the cost aspect.
I tried to evaluate the program on its abilities, not on its cost effectiveness. It is up to a company to asses whether it is useful or not to buy this program.
All in all I have to admit I am pleasantly surprised by the Robotics software, and its usefulness.

## 7.0.0. Literature

[1] BUILD User Guide by McDonnell Douglas
Manufacturing & Engineering Systems Company

[2] PLACE User Guide by McDonnell Douglas
Manufacturing & Engineering Systems Company

[3] COMMAND User Guide by McDonnell Douglas
Manufacturing & Engineering Systems Company

[4] Robot Manipulators by Richard P. Paul
ISBN 0-262-16082-X

# Appendix A: BUILD

## A.0.0.0. INDEX

# Appendix A: BUILD: A functional overview.

## A.1.0.0. BUILD Concepts

### A.1.1.0. BUILD File Types

There are three file type which are created by BUILD. Some of these are the file types used by SIMULATION (as well as COMMAND and ADJUST) to describe a robot or device.

1. BUILD File (.BLD on VAX/VMS and UNIX). The BUILD file contains the basic device description. It essentially keeps a record of all the information which has been entered into BUILD. This file is an input file as well as an output file. When used as an input file, BUILD may be used to edit the parameters of an existing device or BUILD may simply be used as a means of displaying the device parameters. The BUILD file is not used by SIMULATION, COMMAND or ADJUST.
2. DEVICE File (.DEV on VAX/VMS and UNIX). The device file contains the connection tree which describes the device along with the link names and the associated part names. The Device file points to the Device Control Information file. The Device file is created by BUILD. When a cell is saved in SIMULATION the information in the Device file is transferred to the Cell file.
3. DEVICE CONTROL INFORMATION File (.DCI on VAX/VMS and UNIX). The Device Control Information file defines device characteristics for the device. Such information as the kinematic attributes of the device, the allowable motion modes, the maximum joint speeds and acceleration, and more are stored in this file. This file is created by BUILD. SIMULATION, COMMAND, and ADJUST obtain their kinematic information from this file. This file is not in a human readable format.

The following file types are referenced to by BUILD but not actually used by BUILD.

1. PART File (.PAR on VAX/VMS and UNIX). The Part file contains the graphical display data for each link of the device. BUILD only references to the names of these files. It does not actually read the files. The Part file is created by the UNIGRAPHICS to SIMULATION or the IGES to SIMULATION utilities the SIMULATION.
2. COORDINATE SYSTEM INFORMATION File (.CRD on VAX/VMS and UNIX) Coordinate System Information define the attributes of particular "coordinate systems" that are used to represent robot arm positions, tool tip position constraints, etc. CRD files are also used for Dependent Joint mapping and Similar Device Kinematics mapping. Coordinate System information files are associated with the device by using BUILD. These files are defined by the user, using a UNIX text editor such as the vi-editor. BUILD only references to these files, it doesn't actually read them.

A.1.2.0. Data required for Device Modelling

Before a user can model a new device, a certain amount of information must be obtained. In many cases, this information may be obtained from the manufacturer of the robot or the device. Not all information may be required for all devices. If a device is being modeled for simulation only, many of the parameters which affect off-line programming will not be needed. Keep in mind that the more accurate the model needs to be the more accurate the information needs to be.

1. Device Drawings. The user should have access to accurate drawings of the device being modeled. These drawing are needed to created the CAD representation of the device.
2. Link Dimensions. The translational and rotational offsets between each joint.
3. Joint Data. The motion for each joint must be known, this consists of the following data:
   a) Type of Joint. A joint may either be rotational or translational. If a joint is both, for instance a screw, two separate joint will have to be defined and later connected with the dependent joint function.
   b) Joint Limits. The range of each joint.
   c) Joint Dependencies. Does the motion of one joint affect the limits of the other joint?
   d) Joint Speeds. The maximum joint speeds.
   e) Joint Accelerations. The acceleration rate associated with the maximum joint speeds.
   f) Home Position. The initial position of each joint.
4. Allowable Arm Configurations. How does the controller handle situations in which the device can reach the same position with multiple joint solutions?
5. Allowable Motion Modes. The types of motion the controller supports.
6. Default Motion Mode. The default motion mode used by the controller.
7. Allowable Tool Points. Tool positions or orientations which are not allowed should be defined.
8. Tool Speed. The maximum allowable tool speed.
9. Tool Acceleration. The maximum allowable acceleration and deceleration of the tool.
10. Coordinate Systems.Definitions of the coordinate system used to program the device. This information is only needed if the device will be programmed off-line by COMMAND.
11. Device Absolute Coordinate System. The location of the origin of the coordinate system used to program the device. This information may be less important for some devices, for example, devices which are controlled with joint values. Again this information is only necessary for off-line programming.
12. Inverse Kinematics Algorithm. If the user plans to define a device which will not be supported by the standard inverse kinematics algorithm, the user will need to supply BUILD and SIMULATION with an inverse kinematics algorithm which will support the device. The user will have to write a program which will interface with SIMULATION and BUILD and will solve the inverse kinematics of the device.

A.1.3.0. Part File Creation for BUILD

The creation of a device in BUILD is accomplished by specifying a series of transformations which describe the relationships between the links of the device. The geometry for each link must be created on a CAD system ( for instance UNIGRAPHICS II) as a separate part. Each of these parts is modeled in a certain position relative to the absolute coordinate system. BUILD assumes that each part is modeled such that the axis of motion of the link is located at the absolute coordinate system. This allows the motion of each link to be defined as a rotation about or translation along its absolute coordinate system.
BUILD is used to define the relationship between the absolute coordinate systems of each consecutive link.

A.1.4.0. Device Kinematics for BUILD

Once the location of each link coordinate system is known, the transformations between these coordinate systems may be defined. This series of transformations, from the base of the device to its faceplate, define the forward kinematics of the device. Given the joint values, forward- or direct kinematics is the process capable of describing the position of the faceplate or tool of a device in relation to its base.

The user must determine the relationship between each consecutive link coordinate system. This relationship must be described as a series of constant and/or variable transformations. A constant transformation describes either a translational or rotational offset that never changes, regardless of positioning of the joints of the device. This definition would include such values as link lengths, joint offsets, or constant angular offsets. A variable transformation defines the motion of a joint. It is called variable because the amount of the transformation changes as the device moves through its motions. Variable transformations may either be translations (prismatic joint) or rotations (revolute joint). A third type of transformation is called a dependent transformation. A dependent joint is a special kind of variable transformation. The amount of this transformation is derived from the value of some other joints within the device. Since the joint value is not independent of other joints, it does not add a degree of freedom.

All transformations are defined using the right-hand rule.

BUILD is used to define a series of transformations which define the relationships between each successive link coordinate system, starting at the base. The user must define the transformations which describe how to get from one link coordinate system to the next. These transformations show how to completely align the coordinate system on one link with the coordinate system of the next link.

All constant transformations must have a specified amount of translation or rotation. If the transformation is a translation, the amount may be expressed in either units of inches or millimeters. Build allows the user to change units at any time. If the transformation is a rotation, the amount must be expressed in degrees.

All variable transformations must have special joint information supplied. This includes the name of the joint, the constraints or limits on the motion of the joint, the joint's maximum speed and acceleration, and the joint's home position. The same rules for units which apply to constant transformations also apply to variable joint information.

A.1.5.0. Joint Constraints

For each variable transformation or joint a joint-limit must be specified. These constraints define the working range of each joint. These values are used by SIMULATION to compute the joint limit percentages which are shown on the Joints display. These values will also affect the Joint alarm feature of Simulation.

A.1.5.1. Constraints as functions of other joints:

While limits of the type described above will accurately define the constraints on most joints, other joints constraints are more complicated. The limits of some joints change as the position of other joints in the device change. This type of constraint may be due to couple linkages or gear ratios.

In addition to the normal high and low joint values, BUILD allows the user to define an equation which describes the joint limits as a function of some other joint values. If a robot has a coupled drive linkage such as a parallelogram, it most likely needs to make use of this type of joint constraint.

## A.2.0.0. Device Types suitable for BUILD

BUILD will accommodate the inverse kinematics of most of the current commercially available robots. However, BUILD users should be familiar with the basic requirements a device must meet to be analyzed by BUILD. It is possible to simulate devices which do not use the standard inverse kinematics, however, it is than necessary for the user to write the custom kinematics algorithms.

### A.2.1.0. Open Loop Mechanism

The device must be an open loop mechanism. This means that one end of the device must be fixed while the other end of the device is free to move in a 3-D environment. It is on this free end that the Working Tpoint will be created. On a robot, this free end is the faceplate or tool tip.
It is possible to create some Closed Loop Mechanisms by using the Dependent Device feature in SIMULATION. The mechanism is divided into smaller devices. Generally, there is one device for each part of the mechanism which is connected to the ground. The mechanism is then assembled in SIMULATION in such a way that the motion of each device is dependent on some other device.

### A.2.2.0. Device Requirements

In order for a device to use the kinematic algorithms automatically generated by BUILD, the device must have no more than 6 degrees of freedom, and must be built of various components:

1. The arm of the device may be composed of up to three joints that may be revolute (rotational) or prismatic (translational).
2. The wrist of the device may be made up of no more than three joints which must be revolute.
3. The main function of the arm is to position the end of the device while the wrist is used in attaining the correct orientation.
4. Generally, the wrist joints must occur after the arm joints. The exception to this rule occurs when the last arm joint is prismatic, the first wrist joint is revolute and both move along the same axis.
5. A 3-degree-of-freedom arm must consist of no more than two joints that move in one plane. The remaining joint either rotates or translates that plane into a third dimension. One restriction is that this plane must not be perpendicular to the x axis of the world or base coordinate system. That is, this plane must not move in the YZ plane.
6. The wrist joint axis may intersect at one or two points. If They intersect at one point, any combination of wrist joints may be used. If the wrist joint axes intersect at two points, then the first wrist segment must move in the same plane as the arm joints as described above. Generally, the wrist should be defined in such a way that the axis pointing out of the end of the device is the X-axis.

A rule of thumb is that if the last segment of the wrist and one portion of the arm is ignored, the rest of the device must lie in one plane or several parallel planes.

A.2.2.1. Offset Wrist Robots:

Some six axis robots which have offsets between the wrist axes may be modeled by BUILD even though they don't adhere to rule 6. These robots cannot make use of the exact kinematic algorithm used by SIMULATION. These offset wrist robot will be handled by a special iterative kinematic algorithm if they are defined as follows. Joint 4 (the first wrist axis) must be a rotation about the x-axis. The next joint must be a rotation about the same axis as the offset. The last joint must be defined as a rotation about the x-axis.

A.2.2.2. Devices with no Inverse Kinematics:

If a device does not meet the above requirements, it can still be simulated in SIMULATION. Unless another source for the kinematics is supplied, these devices will not have the required "Inverse Kinematics" parameters. BUILD will warn the user in the event of such a situation. This type of device may be simulated in SIMULATION but in a very limited fashion.

A.2.3.0. Dependent Joints

BUILD allows devices to be created which contain joints whose values are dependent on other joints within the device. A dependent joint should be defined when the motion of a single degree of freedom is performed by more than one physical joint. A dependent joint does not add an additional degree of freedom to the device. No joint limit or velocity limit check is performed on dependent joints. In addition a dependent joint may not be controlled independently by the Goto Joints command in SIMULATION.
Because of the great number of functional relationships which may be defined for dependent joints, BUILD will not attempt to derive an inverse kinematics algorithm for such devices. Either an external kinematics program or similar device kinematics must be used for dependent joint devices.

A simple example of the use of dependent joints is in the modelling of a telescoping joint. The telescoping joint performs motion in only one direction, but is actually composed of three segments which move together in the same direction. Each segment moves one third of the total distance of the move. This system would be modeled in BUILD by adding two dependent joints following the variable joint transformation. The two dependent joints represent the last two segments of the telescoping joint. The values of these joints are functions of the value of the total length of the move. The all adhere to the rule:

local translation = (total translation)/3

A Coordinate system File is necessary to define the above equation which will map the set of joint values which SIMULATION computes to the joint values used to display the graphics. The CRD file must cause the joint values to be output in units of inches and radians. The use of any other units will cause incorrect motion simulation. The SIMULATION Joints Display will only show the limit percentage of one joint. If the first joint exceeds a limit, the other two joints, by definition,will also exceed their limits. If an interactive Goto Joints command is attempted only the first joint may be controlled, but all three joints will move.

## A.3.0.0. Sources of Inverse Kinematics

Inverse Kinematics is the process used to convert a position and orientation (i.e. a tpoint) into joint values of a device. Although SIMULATION uses built-in algorithms that will support most robots, it is possible to obtain the kinematics analysis form other (external) sources.

### A.3.1.0 Devices with standard BUILD Kinematics

The normal mode for device modelling is to use BUILD to automatically generate the inverse kinematic parameters to be used by the SIMULATION kinematic analyzer. The device must adhere to the rules listed in section A.2.2.0. for BUILD to be able to generate these parameters.

### A.3.2.0. Devices with no Inverse Kinematics

If BUILD for some reason, can not generate the inverse kinematic parameters for a particular device, the SIMULATION kinematic analyzer will not be able to perform the inverse kinematics. Such a device may be simulated in only a very limited manner. A device without inverse kinematics cannot be directed to move to a tpoint. For this reason the following commands will not be supported in SIMULATION: Goto Tpoint, Goto Position, Goto Circle, and Define Dependent Device. This leaves only Goto Joints and Goto Home as supported motion specifiers. There is, however, a restriction on their use. These commands can only be used with Joint Interpolated or Slew Motion, as Straight Line Motion requires inverse kinematics. Coordinated Motion has the same restrictions as single device motion, only Goto Joints or Goto Home is supported in Joint Interpolated or Slew Motion Mode.

### A.3.3.0. Devices with External Inverse Kinematics

A device which obtains its inverse kinematics algorithms from an external program, may be defined. When this option is used, the inverse kinematics algorithms which BUILD generates automatically will not be used. This capability may be useful when the SIMULATION kinematic analyzer does not give the desired results. For example, this may be required when a particular robot handles singularities in a non standard way, or when a four axis device does not align properly. An external kinematics program may be the only way to simulate some robots which BUILD would not normally handle, for example, a seven axis robot.
The external program must be able to solve the inverse kinematics of the device in question. This means that it should be able to compute joint angles or joint displacements when given a position and orientation with which to align.
SIMULATION and BUILD communicate with the external program by sending and receiving message blocks of packed data.

A.3.4.0. Similar Device Kinematics

There is one other source for the inverse kinematics algorithms for a particular device. Similar Device Kinematics allows the inverse kinematics for a device to come from the definition of a different device. During a simulation, whenever SIMULATION needs an inverse kinematics solution, it will use the kinematics solution (either internal or external) defined for the similar device. The results of the inverse kinematics are then mapped to a new set of joint values via a coordinate system file. All joint limits and velocities will be checked against the new set of values.

The automatic kinematic algorithms derived by BUILD occasionally do not yield the desired alignment results for devices with less than six degrees of freedom. In many of these cases, it is possible to add joints to the definition of the device which will cause it to work properly (a three axes device may be modeled as a six axes device). Simulation will be able to compute the desired joints and a coordinate system mapping will remove the unwanted joints. The coordinate system mapping must output joint values in units of inches and radians. The use of any other units will result in incorrect motion simulation.

Similar device kinematics are also useful when dependent joints are used. BUILD will not generate an automatic similar device algorithm when a device has at least one dependent joint. The inverse kinematics for such a device can sometimes be obtained from a similar device which does not have dependent joints. Figure A.1 gives an overview of the flow of Joint Value Data.

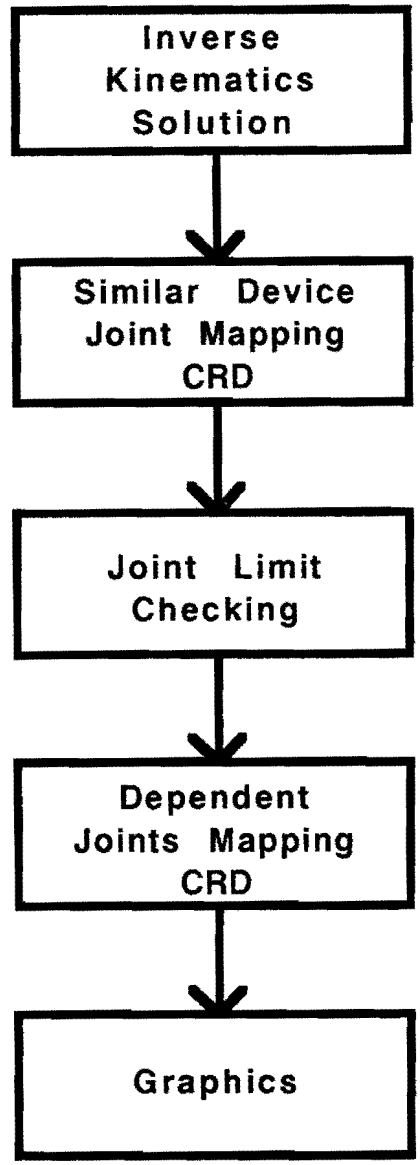


Fig A.1: Flow of Joint Value Data.

# Appendix B: SIMULATION

## B.0.0.0. INDEX

# Appendix B: Simulation: A functional overview.

## B.1.0.0. SIMULATION Concepts

### B.1.1.0. The SIMULATION Database

SIMULATION maintains a database with the following major "file entity" types:

1. Cell
2. Device
3. Sequence
4. Part
5. Device Control Information
6. Coordinate System Information
7. IGES
8. Timing Data

In addition to the above major "file entity" types, SIMULATION maintains the following data entities associated with each cell:

1. Frame
2. Tpoint
3. Connection Tree

B.1.1.1. Data base entity naming conventions:

All SIMULATION database entity names follow the same naming convention except for the tpoint data entity. The tpoint naming convention will be discussed at the end of this section. A SIMULATION entity name is defined to be one to nine alphanumeric characters long. The first character is an alpha character from A to Z. Lower-case characters may be used but SIMULATION will convert them into uppercase.

All SIMULATION file entity types and cell entity types follow the SIMULATION naming convention. In addition , following items also adhere to this naming convention:

1. Operation Names
2. Segment Names
3. Variable Names

The tpoint naming is different than all other names in that it is composed of two names: the tpoint father frame name, followed by the tpoint name. The frame qualifier on a tpoint name is required because tpoint names by themselves are not unique within a cell. There may be two tpoints called TP1. But there can only be one tpoint named TP1 connected to any frame.

B.1.1.2. File entity type descriptions:

1. Cell. A SIMULATION cell contains all the information related to a cell, including the frames in the cell, the connectivity relationships among the frames, the parts associated with the frames and the tpoints connected to the frames. The cell also contains pointers to Device Control Information file entities for each device in the cell. Cells are created by the user using cell editing commands. Cells can be saved for later recovery.
2. Device. A SIMULATION device is a special kind of cell that contains the connection tree to describe a particular device, as well as part, frame, and tpoint information associated with the device. The device points to a Device Control Information file entity for that device. A user can create a new device with the BUILD program. The devices provided by McDonnell Douglas reside in the SIMULATION system library.
3. Sequence. A SIMULATION sequence consists of a series of statements that control the execution of a SIMULATION working session. Sequences are created by the user during an edit sequence session. Most of the SIMULATION interactive functions have a corresponding sequence statement.
4. Part - A SIMULATION part contains the geometric definition of a single rigid body. Part geometry is defined with a CAD-modeller such as UNIGRAPHICS II. SIMULATION parts are created by converting CAD defined models into SIMULATION parts using the following geometry conversion utility: UNIGRAPHICS II to SIMULATION. A SIMULATION part may contain only splines, arcs, lines, and tpoints (coordinate systems). The spline type is an open periodic cubic spline, which is the same spline type the UNIGRAPHICS II uses. The UNIGRAPHICS II Geometry Conversion Utility in SIMULATION can transfer the spline data between UNIGRAPHICS and SIMULATION. Parts are associated with SIMULATION frames using the SIMULATION "Create Frame" function. A frame may have no part or one part associated with it.
5. Device Control Information - The Device Control Information file entity defines device characteristics for a particular device. Such information as the kinematic attributes of the device, the allowable motion modes for the device, and the maximum joint speed allowable for the device are stored in the Device Control Information file entity. Device Control Information file entities are created with the BUILD program.
6. Coordinate System Information. Coordinate System Information file entities define the attributes of particular "coordinate systems" that are used to represent robot arm positions, tool tip positions constraints, etc. Coordinate System Information files are associated with the devices and are defined by the user using a text editor. Coordinate System Information file entities are associated with Device Control Information file entities using the BUILD program.
7. IGES. IGES file entities are the output from the SIMULATION to IGES geometry conversion utility and the input to the IGES to SIMULATION geometry conversion utility. IGES file entities can also be created by some CAD modelling systems such as UNIGRAPHICS. SIMULATION supports the IGES version 2.0 format.

8. Timing Data. A timing file contains the parameters for the timing model of a robot. The timing model accounts for acceleration and deceleration during robot moves. The timing file is generated by optional Cycle Time Analyzer software, which is not a standard part of ROBOTICS software. However, CTA (Cycle Time Analyzer) is available at the site in Eindhoven, University of Technology. See also section B.2.0.0. Motion Simulation.

B.1.1.3. Cell entity type description:

1. Frame. Frames are the major building blocks for SIMULATION cells. A frame has the following information associated with it:

   a) One or no parts.
   b) One connection to a father-frame and a matrix that relates the position of the frame to its father.
   c) May or may not have connection(s) to son frames.
   d) May or may not have connection(s) to tpoints.
   e) A colour assigned to its part.
   f) A colour assigned to its tpoints.
   g) A display tolerance.

2. Tpoints. Tpoints are the positioning entities in a SIMULATION cell. A tpoint is a coordinate system that fully defines six degrees of freedom in space.

3. Connection Tree. The connection tree reveals the connection of the frames to each other and to the world frame. See section B.1.3.0. for more details.

B.1.2.0. The SIMULATION User and System Libraries

SIMULATION file entities may reside in either one or more SIMULATION user libraries or in the SIMULATION system library.

The allocation of SIMULATION user libraries is dependent on the machine implementation of SIMULATION; but in the case of the Eindhoven University of Technology, user libraries are physical directories associated with a validated user logon. When one creates a sequence, saves a cell or a part, they are stored in ones user library.

The SIMULATION system library was created when the SIMULATION software was installed. All SIMULATION devices installed with SIMULATION and their associated Device Control Information, Coordinate System Information and Part file entities are stored in the SIMULATION system library. In addition to devices, certain cells and parts provided by McDonnell Douglas for its tutorials and installation checkout are also stored in the system library.

SIMULATION functions that need to access file entities use a "file entity search" hierarchy. SIMULATION uses a well-defined set of rules to determine where the file entity exists and whether it should be loaded from the user library or the system library. These rules are:

1. The user library is searched first for the specific file entity.
2. If the file entity does not exist on the user library, then the system library is searched.
3. If the file entity is not found on the system library, the user is informed that the file entity does not exist.
4. If the file entity exists on both the user and the system libraries, SIMULATION will use the file entity from the user library.

To access file entities from another user library the "Change Search Directory" function is available.

B.1.3.0. The Connection Tree

The concept of a connection tree is useful in describing and communicating information about the frames in a cell and their connectivity relationships.

When a frame is connected to another frame, the connected frame is called a "son" frame, and the frame that the son is connected to is called the frame's "father".

Connectivity relationships represented by connection trees follow certain rules:

1. Only the WORLD frame has no father.
2. A frame can have any number of sons.
3. When a frame moves, all descendants of that frame move.
4. When a frame moves, no ancestors of that frame move.

Tpoints have connectivity relationships to frame that can also be represented in a connection tree, but tpoints follow slightly different connectivity rules:

A tpoint can have no sons frames or tpoints; in other words, a tpoint can only be a "leaf" in a connection tree.

The SIMULATION function "Display Connection Tree", will display the connection tree for the current working cell. A textual method is used to represent the connection tree, indentation is used to illustrate connectivity relationships. See figure B.1 for an example.

```
0    WORLD:AXIS
1        KUKAWPA00:KUKA00
2            KUKAWPA01:KUKA01
3                KUKAWPA02:KUKA02
4                    KUKAWPA03:KUKA03
5                        KUKAWPA04:KUKA04
6                            KUKAWPA05:KUKA05
7                                KUKAWPA06:KUKA06
1        MANWPA0:MANXZX00
2            MANWPA1:MANXZX01
3                MANWPA2:MANXZX02
4                    MANWPA3:
```

Figure B.1: An example of a connection tree.

### B.2.0.0. Motion Simulation

During SIMULATION interactive and sequence "Goto" and "Move Frame" functions, the SIMULATION software is controlling the simulation of motion that appears on the graphics display. The simulator determines how often new positions of the robot arms and moving workpieces are computed and how often these new positions are displayed. The parameter that determines how often new positions are recomputed is called the "simulation interval". The parameter that determines how often new positions are displayed is called the "display interval". By default, both the simulation interval and display interval parameters are set to 1 second.

The amount of time required to display N arm positions is dependent on many factors. It is dependent on the simulation interval, the display interval, the load on the computer, and it is dependent on how many robots and moving frames are being displayed.
The display time interval is not a real time interval. It is true, however, that the smaller the display interval and or the simulation interval, the slower the motion animation will appear. If the display interval is set lower than the simulation interval, the simulation interval is automatically reduced to be equal to the display interval. If the display interval is set higher than the simulation interval, the simulation interval remains unchanged.
The simulation interval can be changed with the "Set Simulation Interval" function; The display interval can be changed with the "Set display Interval" function.
Both intervals will be affected by the "Faster" and "Slower" functions. The "Faster" button will have the effect of doubling both intervals, the "Slower" button will halve both the simulation and display interval.

If real-time synchronization is on, then the time SIMULATION takes to simulate robot motion more closely approximates the actual real time for that motion. How close the approximations is depends on whether a timing file (created by the Cycle Time Analyzer software) is being used to compute the cycle times for the robot motion.

## B.3.0.0. SIMULATION Informational Displays

In addition to the graphical data representing the cell components, the monitor also has a maximum of seven informational displays relating to the cell simulation.
Following are these seven informational displays:

1. The "MENU" Display
2. The "MOVE TEXT" display
3. The "COLOUR PART/TPOINT" display
4. The "STOPWATCH" display
5. The "SEQUENCE" Display
6. The "DIALS" Display
7. The "JOINTS" Display

Most of these informational displays may be optionally removed from the monitor with the "Blank Text" function and can be redisplayed with the "Unblank Text" function.

### B.3.1.0. The "MENU" display

Through the "MENU" display SIMULATION communicates with the user, the user is confronted with a menu which either lets him execute a function or select another menu. During execution of a function the user is informed of the status of that SIMULATION function through the menu display. For more details see the Using SIMULATION menus section, section B.4.0.0.

### B.3.2.0. The "MOVE TEXT" display.

The "MOVE TEXT" display is used to indicate the position of a frame or tpoint during a "Move Frame" or "Move Tpoint" function. The display illustrates the position of the moving frame or tpoint with respect to the "father" frame of the frame or tpoint. The translational information is the distance the moving frame is from its father, the rotational information consists of the angels in degrees that the frame is rotated with respect to its father.
The "Set Distance Units" function allows one to work in either millimeters or inches. The current distance unit is displayed to the right of the "TRANSLATION" header, no units displayed indicates inch units.

### B.3.3.0. The "COLOUR PART/TPOINT" display

The "COLOUR PART/TPOINT" display is displayed if the function "COLOUR PART/TPOINTS" is invoke, it lets the user select the colour for a frame, or the tpoints connected to a frame. A frame may have only one colour, the tpoints connected to a frame have only one colour. The default colour is White.

### B.3.4.0. The "STOPWATCH" display

The "STOPWATCH" displays the estimate cycle time that the SIMULATION simulator computes for a series of SIMULATION robot motion commands and sequence Move Frame commands. The SIMULATION commands that can affect the stopwatch value are interactive. The stopwatch units are minutes to the left of the ":", and seconds to the right of the ":".

B.3.5.0. The "SEQUENCE" display

The "Sequence" display is used to display the current sequences. See the "Sequence and sequence editing" section (B.5.0.0.) for more details.

B.3.6.0. The "DIALS" display

The "DIALS" display, displays the functions the HP-9000 dials have at the current state in the program.

B.3.7.0. The "JOINTS" display

During robot motion, the "JOINTS" display gives a continuous readout of the state of the joint values for each of the devices in the cell. The limit on the number of devices which may be displayed is a function of the joint display mode and the number of joints in the devices. The state of a joint is a percentage that represents how far the joint is from its limit. If a joint exceeds its limit, its reading will exceed 100% by the corresponding amount. The joint display will indicate a joint limit error by placing the character "j" next to the percentage of the joint with the error.
The Joints display will also signal the user when a joint velocity limit has been exceeded. If a joint exceeds its maximum velocity, the character "v" will be displayed next to the percentage of the corresponding joint. The joint velocity checking may be turned off with the "Device Joint Velocity Check On/Off" function.
In addition to the current state of the joint values, the joints display has a second column of information that displays the maximum values for the joints since the last "Reset Joint Data" function was executed.

The joints display will indicate the current active device. An arrow (->) will point to the active device.

The current state of alignment of the active device with respect to the last commanded goto is also represented on the JOINTS display. The state of alignment is at the end of the joints display and will be blank if the device is aligned with the goto tpoint. The alignment state will say "NOT ALIGNED" otherwise. The alignment state reflects the state of alignment at the end of a goto tpoint. The alignment state will be blank while a goto tpoint is in progress unless the device cannot follow the prescribed path.

If a device passes through a singularity position, the joint display will read "SINGULARITY". A singularity is a position with which the device can align with an infinite number of joint solutions. Because all robot controllers treat these cases differently, singularities should be avoided if possible. If it is not possible to avoid a singularities, the user should take extreme caution.

The SIMULATION "Set Joint Display Mode" function is used to change the type of information displayed in the joints display. In addition to displaying only joint percentages, location values may be displayed along with the percentages. These locations values may be joint angels, cartesian coordinates, or some other device specific values.

The Joints display may also be set to only display the "worst" current joint angle percentage and the "worst" maximum joint angle percentage. This abbreviated display should be used when the joint display of several devices is needed or when the animation rate is to be maximized.
The time needed to update a full joints display is NOT negligible, especially if there is more than one device in the cell. If a high animation rate is essential, the user should consider turning off the joints display. This may be done either with the "Set Devices Monitored" function, or the "Blank" function.
The joints display may also be used to show the value of any user defined ports on a device.The name of the port along with its current value is shown at the end of the joints display.
The position and layout of some of the most used information windows can be found in the photo underneath, taken during a typical working session.

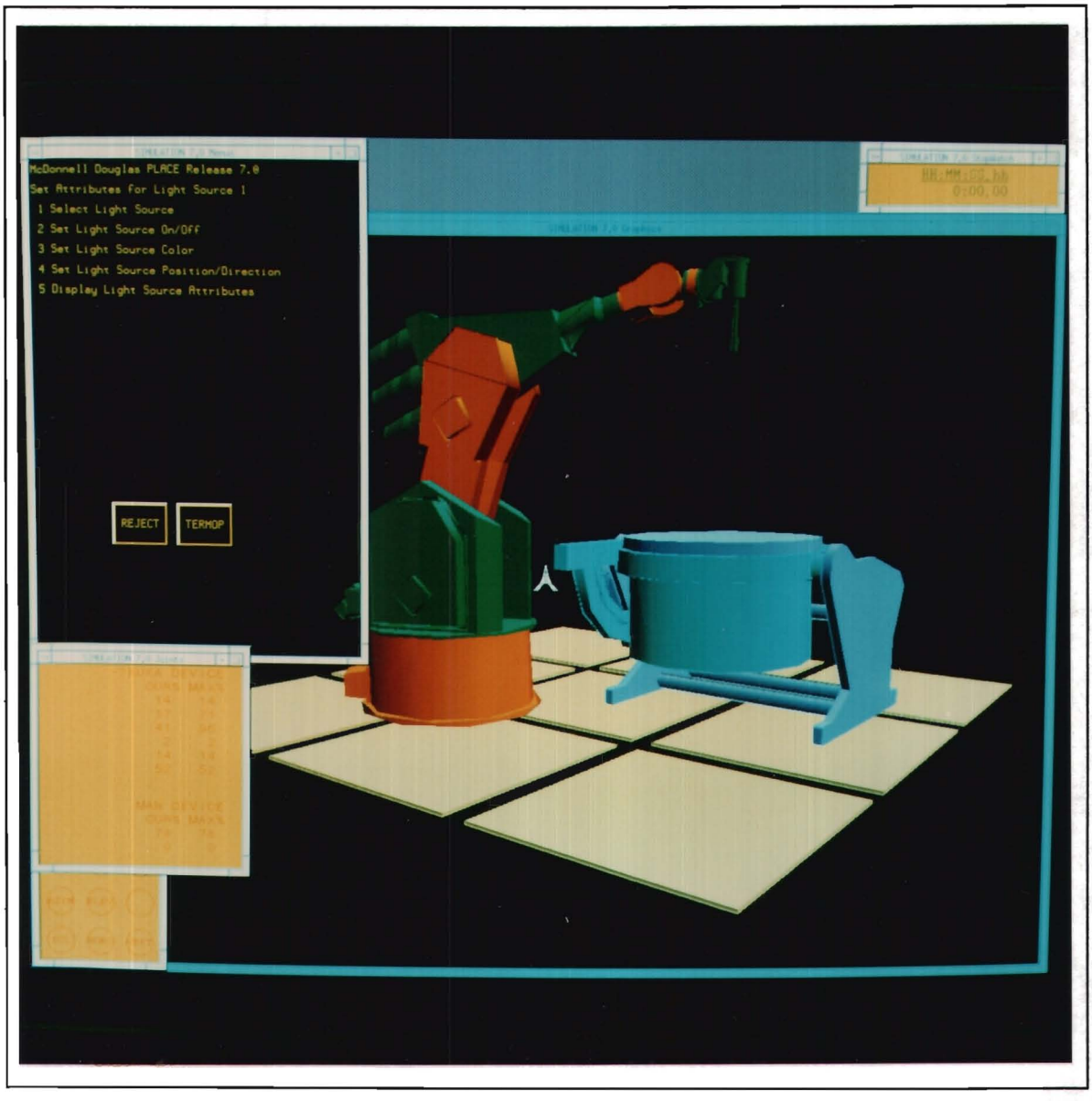


Photo B.1: A typical working session of SIMULATION.

### B.4.0.0. Using SIMULATION Menus

This section explains how to use the SIMULATION hierarchical menu system to select the SIMULATION functions one wants to execute, and how to enter function parameters.

#### B.4.1.0. SIMULATION Menu Types

SIMULATION uses a hierarchical menu system. This means that as one chooses menu options, one is traversing a hierarchy of more and more specific menu options until one has hopefully reached the desired SIMULATION function. When all the parameters for the SIMULATION function are entered, the function will be executed and one will be able to see the result.
There are several types of SIMULATION menus. The first type one encounters is a so-called "choose 1" menu, because one may only choose one option out of a number of options, the Main Menu is an example of such a menu. See figure B.2 for an example of a "choose 1" menu.

```
*****************************************
*                                       *
*    Main Menu                          *
*    1    Cell Editing                  *
*    2    Device Motion Control         *
*    3    Move Frame/Tpoint             *
*    4    I/O and variables             *
*    5    Branching and Conditional Execution *
*    6    View and Display Control      *
*    7    Sequence Editing and Control  *
*    8    Dimensional Analysis          *
*    9    Collision Detection           *
*    10   File Management               *
*    11   Time Control                  *
*    12   File Cell/Terminate           *
*                                       *
*****************************************
```

Figure B.2: The main menu.

A second type of menu is a "data entry" menu, a "data entry" menu is distinguished from a choose 1 menu by the equal (=) signs one sees behind the menu items. The Merge Cell menu is an example of a data entry menu. See figure B.3 for an example of a "data entry" menu.

```
*****************************************
*                                       *
*    Merge Cell                         *
*    1>   Cell Name      =    TEMP      *
*    2    Father Frame   =    WORLD     *
*                                       *
*****************************************
```

Figure B.3: The Merge Cell Menu.

Another type of SIMULATION menu is the "choose n" menu. This kind of menu allows one to choose multiple items from the menu instead of just one. The "List" option under the File Management function uses a menu of the choose n type. See figure B.4 for an example of a "choose n" menu.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                               *
*     List                                                                      *
*     Select Type                                                               *
*     1>    Cell                                                                *
*     2     Device                                                              *
*     3>    Part                                                                *
*     4>    Sequences                                                           *
*     5     IGES                                                                *
*     6     Device Control                                                      *
*     7     All                                                                 *
*                                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure B.4: The list menu.

B.4.2.0. The Function Buttons

On the function keyboard there is a number of buttons which perform a special function in SIMULATION, these button are:

1 ) View Control
2 ) Faster
3 ) Slower
4 ) Interrupt/Resume
5 ) Single Step
6 ) Entry Complete
7 ) Reject
8 ) Terminate Operation

The "View Control" button allows access to the same menus as the main menu item "View Control".

The "Faster" button is used to multiply the display rate and simulation rate by two. The display and simulation intervals control how often the position of robot arms and frames are updated on the graphics monitor during motion simulation.

The "Slower" button has the effect of dividing the display rate and simulation interval by two. See section B.2.0.0. for a more thorough description of the simulation interval and display interval parameters.

The "Interrupt/Resume" button is used after initiating sequence execution with the "Run Sequence", "Edit Sequence", or "Call Sequence" functions to start (Resume) sequence execution. Off course it is also used to interrupt the sequence which is executing.

The "Single Step" button is used to execute sequence statements one at the time. This is especially used when running sequences for the first time making it easy to trace any, if any, mistakes in the sequence being executed.

The "Entry Complete" button is used to indicate the end of a parameter entry. This button will only be displayed (and thus functional) when it has a meaning for the function being executed.

The "Reject" button is used for two purposes: to go back to the previous menu or to cancel the effect of an action just taken. The "Reject" button will perform one or the other, or both of these action, depending on the context of the function.

The "Terminate Operation" button also has two meanings. If one is in the middle of entering parameters for a function and has decided one does not want to execute the function after all, pressing the "Terminate Operation" button will exit the function completely without executing the function at all.
Some functions allow one to execute the function over and over again with different parameters until one presses the "Terminate Operation" button to exit the function.

### B.5.0.0. Sequences and Sequence Editing

A SIMULATION sequence is a stored series of SIMULATION functions that may be played back to simulate a robotic workcell process. Most interactive SIMULATION functions have a corresponding sequence statement. See appendix E for a list of all the valid sequence statements.
During sequence editing, whenever a SIMULATION function is successfully executed, a sequence statement is generated for that function and is written to the sequence output window. The user may see the statements as they are generated in the sequence output window. When a sequence editing session is completed, the contents of the sequence output window may be saved in the user database and later "played back" with the "Run Sequence" function.
Sequences may be used effectively during cell layout to create a workcell process simulation that may be replayed for each new trial layout. When the cell layout is complete, the sequence can be used as an input file to the COMMAND module to generate native mode robot programs for those robot languages that are supported by COMMAND.
The sequence display window contains information about the current sequence being edited with the "Edit Sequence" function or played back with the "Run Sequence" or "Call Sequence" functions. Section B.5.1.0. "The Sequence Display Window", describes the information presented in the display window in some detail for editing and running sequences. Section B.5.2.1. describes how the display window is used to monitor the execution of parallel sequences.

#### B.5.1.0. The Sequence Display Window

The sequence display window is used to display the sequence statements from the sequence currently being edited, run, or called, and to give certain information about the state of sequence execution. The sequence window is located at the bottom of the graphics screen, and may be displayed with the "Display Sequence On/Off" function or automatically displayed when sequence editing is activated.
The sequence display window is composed of three parts:

1. The sequence information line
2. The sequence output window
3. The sequence input Window

The top line of a sequence window is the sequence information line. The output window appears immediately below, followed by the input window. A line in the middle of the sequence display window divides the output window from the input window.

B.5.1.1. The sequence information line:

The sequence information line contains information reflecting the state of sequence execution and or sequence editing. This information is displayed in fields separated by colons (:). Following is a description of these fields from left to right:

1. Sequence execution state. The sequence execution state indicates whether sequence execution is active and, if sequence execution is active, whether a sequence is being edited or just executed. The execution state may be:
   a) "No Active sequence"
   b) "RUN". A sequence is active and has been executed with the "Run Sequence" function.
   c) "EDIT". A sequence is active and has been activated with the "Edit Sequence" function.
   d) "CALL". A sequence is active and has been executed with the "Call Sequence" function.
2. Primary sequence name. When initiating sequence execution, the first sequence that is specified on a "Run Sequence" or on a "Edit Sequence" function is the primary sequence. Any other sequence executed on a "Run Sequence" is called a secondary sequence.
3. Sequence stack depth. Every time a sequence is run without terminating execution of the previous sequence, the new sequence is initiated as the "active sequence" and the previous sequence is saved onto a sequence stack. The number of sequences on the stack is the sequence stack depth. When one creates a brand new sequence with the "Edit Sequence" function, there will be no sequence on the stack because all the sequence statement will be added by the execution of interactive functions.
4. Source of sequence statements added to the output sequence during sequence editing. There may be three sources of sequence statements added to a sequence:
   a) "User Action". User inserted SIMULATION commands
   b) "Paste Buffer". The current contents of the paste buffer. During sequence editing cut and paste functions are supported
   c) "[sequence name]". The sequence that is at the top of the sequence stack.
5. The state of the sequence execution. If sequence statements are being executed from the active sequence, the following two conditions are possible:
   a) "Running". The user pressed the "Interrupt/Resume" button and the sequence is being executed.
   b) "Single Step". The user pressed the "Single Step" button.
   If sequence execution is interrupted, the following three conditions are possible:
   a) "Sequence Stack Empty"
   b) "Paste Buffer Interrupted"
   c) "[sequence] Interrupted"
6. The mode of sequence editing. This field indicates which of the sequence editing functions is active:
   a) "Insert".
   b) "Delete".
   c) "Cut".
7. The current "find" string. This field contains the last string the user specified on the "Find" function.

B.5.1.2. The sequence input window:

The sequence input window displays the sequence statements that are to be executed when the user presses the "Interrupt/Resume" or "Single Step" buttons.

B.5.1.3. The sequence output window:

The sequence output window has a different meaning depending on the state of sequence execution: Edit or Run.
In the "Run" mode, the sequence statements are scrolled from the sequence input window to the sequence output window as soon as they are executed. No sequence statements from a "Called" sequence ever occur in the sequence output window.
In the "Edit" mode, only those sequence statements added to the sequence are scrolled to the sequence output window. These sequence statements may come from the active sequence, from the Paste Buffer, or as a result of the user executing interactive SIMULATION functions.

B.5.2.0. Parallel Sequences

Parallelism provides a general simulation mechanism for showing more than one robot or frame doing independent motion on the screen. For example, two robots can be shown spot welding the two sides of a car body simultaneously.
The low level unit of parallelism is the sequence. Two or more sequences can execute in parallel, but every record within a particular sequence is executed sequentially.
In addition, functions in the I/O and Variables and in the Conditional Execution menus can provide signalling between sequences within a parallel sequence.
Parallelism is a useful simulation tool for:

a) Cycle time determination
b) Visualization of a process (for instance two synchronised robots)
c) Visual collision detection

B.5.2.1. Sequence display window for parallelism:

If the sequence window is on, the input window is dedicated to displaying the records of the sequences being executed in parallel. Thus, any data lines residing in the input window are erased before the Call Sequence function is executed. These lines will be restored after the Call Sequence has completed.
Two lines are dedicated for each sequence that is executing in parallel. If numerous sequences are executing in parallel, the input window size may need enlarging. The two-line window displays the sequence name, the sequence file line number, and the sequence record that is being executed, or is about to be executed, for that sequence.

B.5.2.2. Notes on parallelism:

1. Because of the enormous coding complexity and confusion to the user, editing a parallel sequence is not allowed. All sequences must be edited one at the time using the normal editing procedures and function as they are proscribed by SIMULATION. The sequences can of course be executed in parallel. It is possible the user uses a normal text editor and in that way edits the sequences in a parallel session.
2. A parallel sequence can, however, be interrupted to permit one to issue user level commands before restarting the parallel sequence execution.
3. A sequence can effect the execution of another sequence. This is both a blessing and a curse. The blessing lies in the freedom and extra capability afforded by letting, for example. a sequence change the feed-rate of a robot that is being controlled by another sequence. The curse is that one can get into tremendous trouble with the extra freedom one is given. The problem here is the user not the software.
4. The execution of the Single Step function with regard to parallel sequences is similar to that for sequentially executing sequences.

## B.6.0.0. Robot Motion and Alignment

In most cases the SIMULATION user will pick tpoints, the robot will move to the selected tpoint, and the joint display will indicate whether final and intermediate positions can be reached.
However, some applications require that the SIMULATION users understands HOW the robot aligns with the working tpoint and the goto tpoint. Also how the robot prorates the working tpoint orientation during straight line or circular move is important. The description below indicates the algorithm that SIMULATION uses.

Terminology varies greatly in robotics.
Here, "wrist" refers generally to the last two or three joints of a robot, they are generally used in achieving a prerequired orientation.
"Arm" refers generally to the first three joints, may they be prismatic or rotational, starting at the base of the robot. These first three joints are generally used in reaching a certain position.

### B.6.1.0. Simple Alignment

In going to a tpoint,SIMULATION always tries to align the orientation and the roll vectors of the goto and working tpoint first. It is assumed that the arm will be able to align the x,y,z, coordinates of the goto and the working tpoint. Of course some of the goto tpoints require that the wrist of the robot bends in a peculiar way in order to align orientation and roll vectors. The resulting wrist configuration may then prevent the arm from being able to align x,y,z, coordinates without disconnecting the links in the robot.

### B.6.2.0. Straight Line Motion

During straight line motion, the tool point of the robot is constrained to follow a straight line with respect to the robot base while the working tpoint proceeds towards alignment with the goto tpoint.
Only one equation (a straight line) exists for describing the path of the tool point from start to the end of the move. Unfortunately, an infinite number of algorithms exist for describing how the orientation and roll at the tool point can vary during straight line motion, while still keeping the tool point x,y,z, on a line. Not surprisingly, every robot that can do straight line motion will use a different (and often proprietary) algorithm for moving the orientation and roll to their final values at the goto tpoint.
The algorithm that SIMULATION uses is as follows. SIMULATION computes where the tool and orientation and roll will be at the end of the move. SIMULATION 'knows' where the starting values are. Eulers theorem then states that given two coordinate systems with the same origin but different axes directions, one of the coordinate systems can be rotated into the other by rotating around a line through the origin of both coordinate systems. SIMULATION calculates this angle and line before the move starts. The tool axes are rotated about this "Euler" line. At the start of the move the rotation change is zero, at the end of the move the rotation change has reached the full Euler angle.

This proration algorithm is used for all types of robots. In many cases, the SIMULATION algorithm for wrist proration is close to what the real robot controller will do. However, the user must be careful, some large discrepancies between the real robot and the simulated robot move may occur when very large angular changes in the tool orientation and roll vectors occur. For moves with small tool axes rotation, the SIMULATION algorithms generally will accurately describe the path of the arm and the wrist.

B.6.3.0. Joint Interpolation

During joint interpolated motion, the robot will move all of its joints in such a way that each of the joints starts and ends the motion at the same time. The set of joint values that the robot moves toward are determined at the beginning of the move. If there is more than one joint solution which will cause the robot to align, SIMULATION will attempt to determine one that is within the limits of the joints. If the robot was set to be in Automatic Wrist Configuration mode in BUILD, then SIMULATION will choose the "best" wrist configuration, i.e. the one which will causes the joints to move the smallest angles.

B.6.4.0. Slew Motion

Slew motion is very similar to joint interpolated motion. The only difference is that each joint will move independently at a constant speed. It will accelerate to its highest speed and will continue to travel at this speed until it has to decelerate to reach its destination. Some of the joints may complete their motion before others.

B.6.5.0. Circular Motion

Circular interpolation causes the tool tpoint of the robot to move in a circular path. The path is formed by three points, the current tool point, an intermediate tpoint, and a destination tpoint.
Circular Interpolation has the same problems as straight line motion There is an infinite number of algorithms which could be used to control the tool orientation and roll as it follows the circular path. SIMULATION uses the same algorithms as straight line motion for circular interpolation. The Euler angle as described in the previous section is computed between the beginning tpoint and the ending tpoint and the orientation is prorated evenly throughout the entire move. The orientation of the intermediate tpoint is not taken into consideration.

B.6.6.0. Velocities and Accelerations

The simulated path of a device will use acceleration and velocity information either from a Timing file or from the robot DCI file. The robot will accelerate to the constant velocity and then decelerate to stop at the end of the move. If the continuous path option is On, the device will not decelerate at the end of the move. For either straight line or circular motion the tool tpoint will use the tool tpoint speed and tool tpoint acceleration values to define the motion. For either joint or slew motion the joints will move according to the joint velocities and accelerations. For most robots, the time for acceleration and deceleration may not be visually noticeable to the user. If there is no Timing File or there is acceleration data in the DCI file, SIMULATION will simulate the motion as if there are infinite accelerations and decelerations at the beginning and end of the move.

B.6.7.0. 5-axis Special Case

5-axis robots present a special problem. In general a 5-axis robot cannot align position, orientation and roll vectors. In fact, 5-axis robots cannot always align even just position and orientation vectors. The question for 5-axis moves is not how does a robot reach a given tpoint, but how close can the robot reach.
Just as is the case with a 6-axis robot, SIMULATION tries to align the orientation and roll vectors, while assuming that the x,y,z, coordinates of the working tpoint and goto tpoint can be aligned exactly. However, in the five axis case SIMULATION will select the wrist position that will minimize the misalignment of the orientation and roll vectors of the working tpoint and the goto tpoint. Moreover, the orientation vector alignment is weighted much more heavily than the roll vector alignment. When two or more of the wrist positions yield approximately the same orientation alignment, then the wrist configuration which creates the best roll vector alignment is used.
For the 5-axis robot user, some tips for more effectively using SIMULATION are:

a) Select the ideal goto tpoint orientation vector first. (Remember that alignment may not be possible exactly, but SIMULATION calculates the nearest solution.) Then move the roll vector by rotating the goto tpoint about the orientation vector to determine the best overall robot position for reaching the desired goto tpoint position and orientation.

b) Try generating the goto tpoints very crudely by using the Create Tpoint function in Goto Joints. Then translate these newly created tpoints to the desired location. The robot should be able to align well with the goto orientation, since the working/goto tpoint alignment was perfect where the goto tpoint was created. Of course the translated goto tpoint should be as close as possible its creation location.

B.6.8.0. Devices with no Inverse Kinematics

Another special class of devices are those which do not have a supported inverse kinematics algorithm. Inverse kinematics is the process used by SIMULATION to convert a position and orientation (i.e. a tpoint) into the joint value of the device. Although SIMULATION will use build in algorithms that will support most robots, it is possible to use BUILD to define devices which SIMULATION's inverse kinematic analyzer cannot handler. Build will warn the user in the event of such a situation. Devices without inverse kinematics may be simulated in SIMULATION, but only in a very limited way.
A device without inverse kinematics cannot be directed to move to a tpoint. For this reason the following commands will not be supported in such a situation: Goto Tpoint, Goto Position, Goto Circle, and Define Dependent Device. This leaves only Goto Joints and Goto Home as supported motion specifiers. There is, however, a restriction on their use. These commands must only be used with Joint Interpolated or Slew motion. Straight Line motion requires inverse kinematics and therefore it is not supported. Coordinated Motion has the same restrictions as single device motion, only Goto Joints or Goto Home in Joint Interpolated or Slew motion is supported. If the user attempts to make an unsupported move, an error message will be displayed.

B.6.9.0. Alignment Problems

There are times when there will be problems with the alignment of a device with its goto tpoint. These cases will only occur when using commands such as Goto Tpoint, Goto Circle and Goto Position. These commands instruct the device to go to a certain position and orientation in space. Errors occur when the device cannot align with the desired tpoint or path. In commands like Goto Joint Position or Goto Home, the device is instructed to go to a predefined set of joint values, so the alignment will not cause any trouble.
The Joints Display will signal an alignment error if the device could not align with the goto tpoint at the end of a move or could not follow the desired path during the move. The most common example of this problem is when the goto tpoint is to far away from the base of the device. In such a case the device will generally reach toward the tpoint and place the orientation of its working tpoint to be parallel with the orientation of the goto tpoint. A second example is a device which has less than 6 degrees of freedom, and therefore cannot align with the goto tpoint. A third example is when the beginning tpoint and ending tpoint yield good alignment, but a straight line path between them may pass through an area where alignment is not possible.
Another common alignment problem is caused when a device is instructed to align with a singularity position. This means that there is a infinite number of kinematic solutions which will cause alignment with the goto tpoint. In many cases this is caused when two or more joints become parallel at such a location. Because all robot controllers use different heuristics to select a joint solution, there is the possibility of inaccurate simulation. For This reason singularities should be avoided if possible. The joints display will indicate whenever a device is at or near a singularity.
If a device is performing either straight line or circular motion, the speed setting determines the speed at which the device will follow this path. It is possible that in attempting such a move, one or more of the joints of the device would have to move faster than their maximum joint speed. SIMULATION will warn the user of this situation by placing the character "v" behind the affected joint, in the joints display. Because all robot controllers handle such a cause differently the user should be extremely cautious in perform these moves.

## B.7.0.0. Compound devices

B.7.0.1. Definitions:

Simple Kinematic Device. A simple kinematic device is any device which may be created with BUILD. The device must be open looped (i.e. one fixed end, and one free end), be made up of revolute and or prismatic joints, have no more than six joints. Each simple device is described in its own Device Control Information file.

Compound Device. A compound device is a general term which describes a device composed of two or more simple kinematic devices. Each compound device is a named entity.

Sub-Device. A sub Device is a simple kinematic device which is a part of a compound device.

Coordinated Motion Device. A coordinated motion device is a compound device composed of two or more sub devices, which coordinate in such a way that their motions begin and end at the same time.

Dependent Motion Device. A dependent motion device is a compound device which specifies that the joint solution for one device is dependent on the joint solution of another device. Dependent motion devices allow for the simulation of closed loop mechanisms and the driving linkages of robots.

B.7.1.0. Coordinated Motion Devices

Coordinated motion devices allow for the simulation and programming of more than one device using simultaneous motion. Many robot controllers have the capability to control not only the axes of the robot in question, but also a number of external axes.These other axes might be used to control the robot's position on a track, a positioning table which is used in conjunction with the robot, or perhaps even to control more than six axes of the robot arm.
The real world devices must be definable as separate simple kinematic devices as modeled in SIMULATION. Each device is separated to the extent that each device has its own kinematic description (.DCI-file) and has separate motion parameters. (Speed, motion mode, goto tpoint, etc).
The define coordinated motion device command is used to create the list of sub-devices of which the coordinated motion device will consist. The coordinate motion device may be thought of as a logical definition of the controller, where the sub-devices would normally represent the robot and its external axes.
The coordinated goto command is used to cause all of the sub-devices of the coordinated motion device to move. The motion of each sub-device will be prorated so that each sub-device will start and stop at the same time.

B.7.2.0. Dependent Motion Devices

A dependent motion device is one whose joint solution is always dependent on the position of some other device in the cell. This capability allows for the simulation of closed loop mechanisms an the driving linkages of robots.
A dependent motion device is controlled by giving it a working tpoint and a goto tpoint by using the Define Dependent Device command. The device will work in a manner similar to that of the Tracking option. The working tpoint will always attempt to align with the goto tpoint. Once a dependent motion device has been defined, it may not be made the active device. The user has no control over some of the motion parameters such as speed and motion mode. This restriction occurs because the motion of the device is dependent on that of some other device.
The Disable Dependent Motion Device command is used to convert a dependent motion device back into a "normal" device.
If a dependent motion compound device is defined in the workcell when the save cell function is executed, the definition for that device will be saved. At a later time when the cell is merged, the compound device will be retained.

### B.8.0.0. Simple sensor

Within SIMULATION there is a possibility of simulating simple sensors and their effect on the robots programming. A simple sensor is a device that returns a single value for a sensed property such as distance, proximity, or pressure.
Five simple sensors are provided with SIMULATION. Some sensors return values derived directly from geometric properties, some others just return "random" values within the legal limits of the sensor. The five sensor provided are:

1. A sensor for measuring distance, that returns the distance from the end of the sensor to the nearest tpoint.
2. A proximity sensor that returns a TRUE if an tpoint is sensed within 25.4 millimeters (one inch) of the sensor.
3. A proximity sensor that returns a TRUE if an tpoint is sensed within one meter of the sensor.
4. A pressure sensor which returns a random number.
5. A temperature probe which returns a random number.

If so desired the user can override the automatically generated sensor values by putting the sensor's port into "Manual Receive Mode".

Hein-Jan van Veldhoven
ID: 221176

# Appendix C: COMMAND

**C.0.0.0. INDEX**

# Appendix C: COMMAND: A functional overview.

### C.1.0.0. COMMAND File types

There are seven major file types used and created by COMMAND, namely:

1. The Cell file. (.CEL extension on VAX and HP) The cell file contains all the graphical information related to the SIMULATION cell. The cell file is created by SIMULATION.
2. The Sequence file. (.SEQ extension on VAX and HP) The sequence file is created by using SIMULATION. This file contains a sequence of SIMULATION commands which together with the cel file simulate a certain cell.
3. The User file. (.USR extension on VAX and HP) The user file is created by the user using a normal UNIX text editor such as the vi-editor. This file serves as the base for sequence and program processing as well as a source for commands native to a particular robot controller.
4. The COMMAND Source Program file. (.CSP extension on VAX and HP) The COMMAND source program file contains both robot motion and logic instructions in a robot independent language. This file is created and used by COMMAND.
5. The Source Robot Program file. (.SRC extension on VAX and HP) This file contains all the robot direct commands in the robot's native language. If a robot does not have a native language, this file will contain a language emulating the teach commands.
6. The Robot Program File. (.rfile extension on VAX and HP) This file contains the robot direct commands and can be transported to the robot controller either by means of floppy disk or tape, or by means of a network link.
7. The Error Message File. (.LIS extension on VAX and HP) The error message file contains a list of errors which occurred during translation.

### C.2.0.0. COMMAND Program Flow

Figure 1 shows the flow of data from SIMULATION through COMMAND. The first step in creating a complete robot program requires the creation of a cell and sequence. After that a User file must be generated containing the appropriate sequence access commands and robot specific instructions. Once these steps have been completed the user enters the COMMAND module and is prompted for the names of the Cell and User files. As processing begins COMMAND expands the sequence access commands in order to extract the specified SIMULATION sequence statements. These sequence statements are translated into robot independent commands, and interspersed with robot logic instructions as specified in the user file. This results in a COMMAND source program file containing both robot motion and logic instructions. During this process, COMMAND internally runs the SIMULATION sequences referenced to by the user file and calculates robot positions based on the SIMULATION computations. The command source program output from the COMMAND sequence processing stage is then translated by a specific Robot Program Translator. The Robot Program Translator will produce a complete robot program in the native language of the robot.

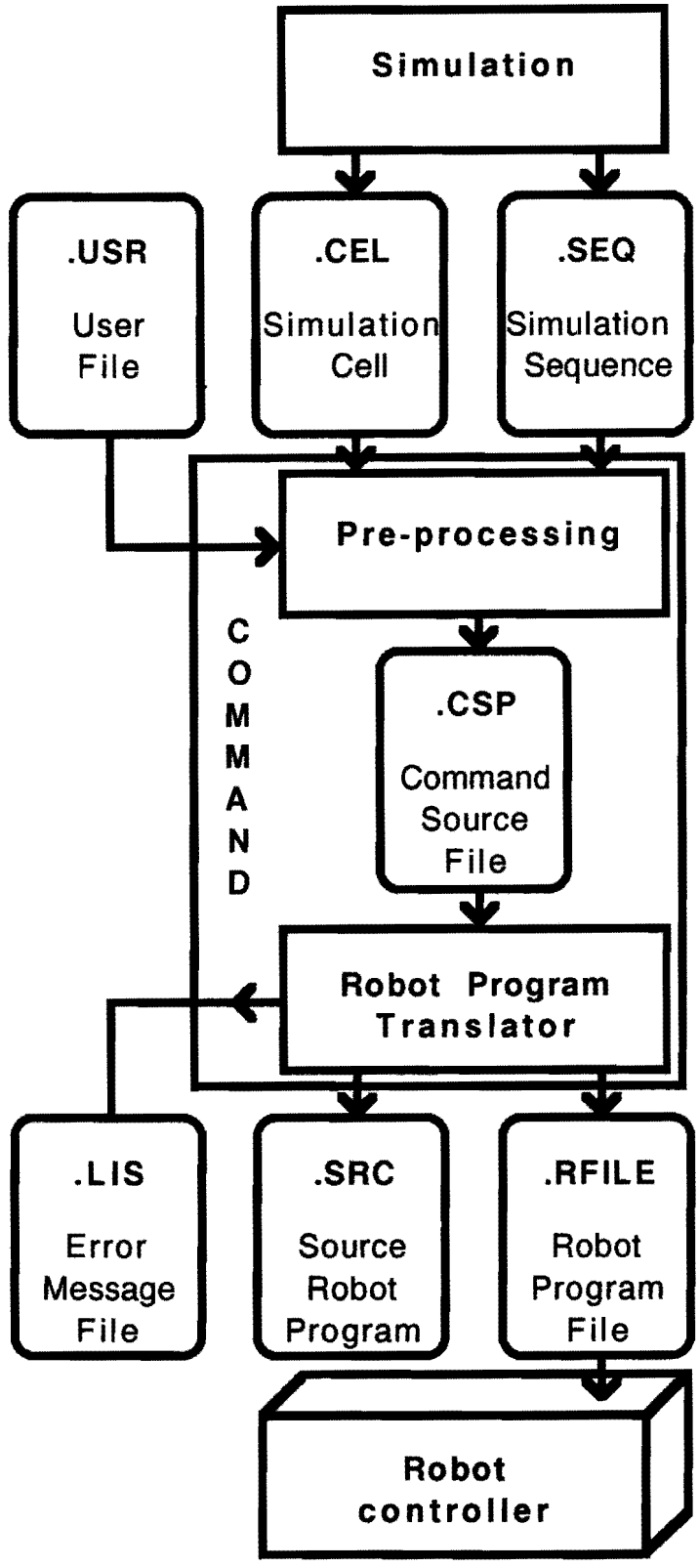


Fig. C.1: COMMAND program flow.

The user may expect to receive three or four types of output (depending on the selected robot) from the translation.First, an error message file will be generated if any errors are found. If COMMAND executes without error a robot program source file and robot program output file may be created. On robots that execute their own "built-in" source language, these two files may be identical. However, on robots that do not have their own language, these two files may be considerably different. The source file for non language driven robots will contain a language emulating the teach commands. The output file will contain the binary dat formatted for a particular robot's controller.
To complete the process, the COMMAND module also provides the user with the capability of transmitting the robot program directly to the robot-controller or outputting it to an appropriate media (i.e. floppy disk or tape cassette). The means and possibilities of communication vary with each translator.

**C.3.0.0. The user file.**

As the user file is the most important file to command, it will be explored a little further. The user file may contain any of the following types of commands:

1. Operation macro. The operational macros allow a user to define robot language commands that will be executed at specific positions identified by a SIMULATION sequence.
2. Sequence Mapping Commands. The sequence mapping commands allow a user to map sequence parameters onto the robot controller's parameters
3. Sequence Access Commands. The sequence access commands alow a user to access and process sequences in a particular manner.
4. Robot Language Commands. Any native robot language command may be input to this file for execution on the robot.

C.3.1.0 Operation Macros

Operation macros are most often used to execute logic and I/O related functions on the robot controller. The most common of these include:

1. Opening and closing grippers.
2. Turning physical sensors on and off.
3. Sending and receiving signals from peripheral equipment.

C.3.2.0. Sequence Mapping Commands

The parameters of the sequence functions, variables, ports, and labels must be mapped to the robot controller for these functions to operate correctly. The mapping for these functions will take place automatically unless the user defines the mapping. The resulting translation will not contain references to the sequence parameters, but to the mapping (robot controller) parameters.

C.3.2.1. Automatic Mapping:

A default list of parameter names for variables, ports, and labels have been defined for each robot translator. As a function is encountered in the sequence during execution of COMMAND, the next available parameter name is pulled from the default list unless, the sequence parameter name is the same as a mapping name and it has not been assigned, then that assignment is made.

C.3.3.0. Sequence Access Commands

Sequence access commands allow the user to include portions of a sequence, an entire sequence, or multiple sequences in a robot program. When a sequence access command is encountered during processing, the appropriate sequence file is opened and becomes known as the active sequence. The processing position in the active sequence is maintained by a sequence pointer. Whenever this pointer is advanced, the sequence commands are processed to update the state of the cell model. Sequence processing takes place in a forward direction only. Therefore, the pointer may be advanced through the sequence but not "backed up" through commands already processed.

C.3.4.0. Robot Language Commands

In a user file it is possible to insert a number of robot direct commands, which can then perform certain functions which might not be incorporated into the SIMULATION sequence commands. This option is often used inside the operation macros. (See section C.3.1.0. for more details).

# Appendix D: SIMULATION and SRCL commands.

**D.0.0.0. INDEX**

# Appendix D: SIMULATION sequence commands to SRCL-commands.

### D.1.0.0. Syntax Rules

| | |
|---|---|
| lower case | indicate syntatic categories |
| ::= | 'to be written as" symbol |
| \| | vertical bar to separate choices |
| < > | choose 1 of the enclosed items |
| [ ] | repeat the enclosed items 0 or more times |
| ( ) | repeat the enclosed items 1 or more times |
| { } | optional items |
| " " | the item appears exactly as shown |
| OTHER ITEMS | terminal symbols |
| NRS | No Resulting Statement |

Frequently Used items:

| | |
|---|---|
| <alpha> | ::=<A\|B\|C\|...\|X\|Y\|Z\|a\|b\|c\|...\|x\|y\|z> |
| <digit> | ::=<0\|1\|2\|3\|4\|5\|6\|7\|8\|9> |
| <name> | ::=<alpha> [<alpha>\|<digit>] -- maximum of nine characters |
| <device> | ::=<name> |
| <framename> | ::=<name> |
| <frame> | ::=<<framename>\|<variable>> |
| <framepair> | ::=<frame>,<frame> |
| <integer> | ::=any integer |
| <operation> | ::=<name> |
| <part> | ::=<name> |
| <port> | ::=<name> |
| <real> | ::=any real number |
| <tpointname> | ::=<name> |
| <tpoint> | ::=<<frame>,<tpointname>\|<variable>> |
| <units> | ::=<"(IN)"\|"MM"> |
| <variable> | ::=<name> |

### D.2.0.0. Program sequence instructions

| SIMULATION command: | SRCL-command (german): |
|---|---|
| JUMP_TO: <label>; | SPG HP<br>AD<br>VZ<br>UP<br>ZY |
| DELAY: <real>; | WRT |
| WAIT_UNTIL: <expression>; | WRT E H/L (Wait until input) |
| IF <expression> THEN JUMP_TO : <label>; | BAW (Under conditions) |
| If <expression> THE EXIT_CURRENT_SEQUENCE:; | BAW (Under conditions) |
| EXIT_CURRENT_SEQUENCE:; | HLT UN<br>Unconditional Halt |
| PAUSE:; | HLT UN |

### D.3.0.0. Movement instructions

| | |
|---|---|
| ACTIVE_DEVICE: <device>; | Initialize instruction set. |
| SET_DEVICE_LOCATION_REPRESENTATION: <crdsys>; | N.R.S. (just one solution possible) |
| SET_DEVICE_CONFIGURATION: <configname>; | **? ? ? ?** |
| SET_DEVICE_MOTION_MODE: <STRAIGHT\|INTERPOLATE\|SLEW>; | LIN/PTP |
| SET_SEQ_MOVE_SPEED: <units>,<real>,<real>; | **? ? ? ?** |
| SET_SEQ_MOVE_ACCEL: <units>,<real>,<real>; | **? ? ? ?** |
| SET_DEVICE_SPEED: <units>,<real>; | GES BAN<br>ALL<br>ACH<br>OV |
| CONTINUOUS_PATH_OFF:; | UES BAN 0%<br>PTP 0 |
| CONTINUOUS_PATH_ON:; | UES BAN 1-100%<br>PTP 1-9 |
| SET_DEVICE_TOOL_TPOINT: <tpoint>; | WZK |
| WORKING_TPOINT: <tpoint>; | WZK |
| FRAME_MAXIMUM_SPEED: <frame>,<real>; | N.R.S./GES BAN<br>ALL<br>ACH<br>OV |
| GOTO_CIRCLE: <tpoint>,<tpoint>,<operation>; | **ZR** |
| GOTO_CRD: (<real>,)<operation>; | LIN<br>PTP |
| GOTO_HOME: <operation>; | LIN<br>PTP |
| GOTO_JOINTS: <units>,<jntval>,<operation>; | LIN<br>PTP |

where:
<configname>::= any valid configuration string (defined by BUILD)

GOTO_POSITION: <frameortpt>,<frameortpt>, LIN
<INC|ABS>,<units>,<posvec>,<rotangs>,<operation>; PTP

GOTO_TPOINT: <tpoint>,<operation>; LIN
PTP

MOVE_ABS: <frameortpt>,<frameortpt>,<frameortpt>, LIN
<rotvec>,<rotamnt>,<units>,<posvec>; PTP

where:

<frameortpt> ::=<<frame>,|<tpoint>>
<rotvec> ::=<real>,<real>,<real>
<rotamnt> ::=<real>
<posvec> ::=<real>,<real>,<real>

MOVE_REL: <frameortpt>,<frameortpt>,<frameortpt>, LIN
<rotvec>,<rotamnt>,<units>,<posvec>; PTP

EXTERNAL_JOINTS_ON: <message>; N.R.S./**ZAC EINS**

EXTERNAL_JOINTS_OFF:; N.R.S./**ZAC AUS**

DEFINE_COORD_MOTION_DEVICE: N.R.S./**ZACEINS**
<crddevice>[,device];

COORDINATED_GOTO: **ZAC EINS**
<crddevice>,(<device>,<gotomode>,)<operation>; **LIZ** (linear)
**ZRZ** (circle)
**PPZ** (point-point)

where:

<crddevice> ::=<name>
<gotomode> ::=<<gotp>|<gojt>|<gohm>|<gocr>|<gopo>|<nomv>>
<gotp> ::=TPOINT,<tpoint>
<gojt> ::=JOINTS,<units>,(<real>,)
<gohm> ::=HOME
<gocr> ::=CIRCLE,<tpoint>,<tpoint>
<gopo> ::=POSITION,<frameortpt>,<frameortpt>,
<INCREMENTAL|ABSOLUTE>,<units>,<posvec>,<rotangs>
<frameortpt> ::=<<frame>,|<tpoint>>
<posvec> ::=<real>,<real>,<real>
<rotang> ::=<real>,<real>,<real>

### D.4.0.0. Text instructions

| | |
|---|---|
| BEGIN_SEGMENT: ; | N.R.S. Used by COMMAND |
| ; <comment> | KOM |
| INTERFERENCE_DETECTION:; | ? ? ? |
| EXAMINE_VARIABLES: <variable>[,variable]; | ? ? ? |
| END_SEGMENT: ; | N.R.S. (Used by COMMAND) |

### D.5.0.0. Memory, variable and calculation instructions

SET_VARIABLES: <variable>,<constant>[, <variable>,<constant>]; LAD Px xxx

DELETE_ALL_VARIABLES:; RS

DELETE_VARIABLES: <variable>[,variable]; RS

CALCULATE: <variable> = <expression>; ARI (UNDER CONDITIONS)

RECEIVE_FROM: {<device>}(,<recvpair>); LAD Px Axx
where:
<recvpair> ::= <port>,<variable>

SEND_TO: (<device>)(,<sendpair>); LAD Ax xxx
where:
<sendpair> ::= <port>,<variable>

RDC: <command>; ROBOT DIRECT COMMAND

**D.6.0.0. SIMULATION instructions with no equivalent SRCL-instruction**
(N.S.R. stands for No Resulting Statement)

ADVANCE_STOPWATCH: <real>; N.R.S.

ALIGN_TPOINTS: <tpoint>,<tpoint>; N.R.S.

ALLOW_COLLISIONS: <framepair>; N.R.S.

BLANK_FRAMES: <frame>[,frame]; N.R.S.

BLANK_PARTS: <frame>[,frame]; N.R.S.

BLANK_SOLIDS: N.R.S.
<SPHERES|BOXES|HULLS|ALL_SOLIDS>;

BLANK_SUBTREE: N.R.S.
<FRAMES|TPOINTS|PARTS>,<frame>;

BLANK_TEXT: N.R.S.
<ALL|(STOPWATCH|JOINTS|SEQWINDOW|MOVETEXT)>;

BLANK_TPOINTS: <frame>[,frame]; N.R.S.

BLANK_UP_TO_WORLD: N.R.S.
<FRAMES|TPOINTS|PARTS>,<frame>;

BOTTOM_VIEW:; N.R.S.

CALL_SEQUENCE: <sequence>[,<sequence>]; N.R.S.

CHANGE_DIRECTORY: <directory>; N.R.S.

CHANGE_DISPLAY_TOLERANCE: <part>,<real>; N.R.S.

CLEAR_CELL:; N.R.S.

COLLISION_DETECTION_METHOD: N.R.S.
<FIXED_INTERVAL|VELOCITY_DISTANCE_BOUND>;

COLLISION_DETECTION_OFF:; N.R.S.

COLLISION_DETECTION_ON:; N.R.S.

COLOR: N.R.S.
<frame>,<PART|TPOINTS>,<color>,<hue>,<sat>,<int>;

CONNECT_FRAMES: <framepair>; N.R.S.

CONNECT_PORTS: {<device>},<port>,{<device>},<port>; N.R.S.

CONNECT_PORTS_EXTERNAL:(<device>),<port>; N.R.S

CONNECT_TPOINT: <tpoint>,<frame>; N.R.S

CREATE_FRAME: N.R.S.
<framename>,<part>,<frame>,<units>,<matrix>;
where:
<matrix> ::=<xvec>,<yvec>,<zvec>,<posvec>
<xvec> ::=<real>,<real>,<real>
<yvec> ::=<real>,<real>,<real>
<zvec> ::=<real>,<real>,<real>
<posvec> ::=<real>,<real>,<real>

CREATE_TPOINT: N.R.S.
<frame>,<tpointname>,<units>,<matrix>;

DEFINE_DEP_MOTION_DEVICE: N.R.S.
<device>,<tpoint>,<tpoint>;

DEFINE_PORT: N.R.S.
{<device>},<port>,<REAL|INTEGER|BOOLEAN>,<INPUT|OUTPUT|INOUTP>;

DEFINE_SOLIDS: <<frame>[,frame]|ALL_FRAMES>; N.R.S

DELETE_DEVICE: <device>; N.R.S.

DELETE_FRAME: <frame>[,frame]; N.R.S.

DELETE_PORT: {<device>},<port>; N.R.S.

DELETE_TPOINT: <tpoint>[,tpoint]; N.R.S.

DEPTH_CLIPPING_OFF:; N.R.S.

DEPTH_CLIPPING_ON:; N.R.S.

DISABLE_DEP_MOTION_DEVICE: <device>; N.R.S.

DISALLOW_COLLISIONS: N.R.S.
<<framepair>[,framepair]|ALL_FRAMES>

DISCONNECT_FRAMES: <frame>[,frame]; N.R.S.

DISCONNECT_PORTS: N.R.S.
{<device>},<port>,{<device>},<port>;

DISPLAY_CONNECTION_TREE:; N.R.S.

DISPLAY_WIREFRAME_FRAMES:; N.R.S

EXTERNAL_PORTS_ON:<message>; N.R.S.

EXTERNAL_PORTS_OFF:; N.R.S.

FASTER:; N.R.S.

FRAME_TO_FRAME: <framepair>; N.R.S.

FRONT_VIEW:; N.R.S.

HIDDEN_LINE_REMOVAL:; N.R.S.

IF <expression> THE CALL_SEQUENCE: <sequence>[,sequence]; N.R.S.

JOINT_ALARM_OFF:; N.R.S.

JOINT_ALARM_ON:; N.R.S.

JOINT_VELOCITY_CHECK_OFF:; N.R.S.

JOINT_VELOCITY_CHECK_ON:; N.R.S.

LEFT_SIDE_VIEW:; N.R.S.

LIST_FRAMES_WITH_SOLIDS:; N.R.S.

MERGE_CEL: <cell>,<frame>; N.R.S.

MERGE_DEVICE: <device>,<frame>; N.R.S.

MOVE_TPOINT_GROUP: <BLENDED|SIMPLE>,<frame>, N.R.S.
{tpointname|variable}, {tpointname|variable},
{tpointname|variable},<rotvec>,<rotamnt>,<units>,<posvec>;

PERSPECTIVE_OFF:; N.R.S.

PERSPECTIVE_ON:; N.R.S

POLYGON_EDGES_OFF:; N.R.S.

POLYGON_EDGES_ON:; N.R.S.

REAL_TIME_SYNC_OFF:; N.R.S.

REAL_TIME_SYNC_ON:; N.R.S.

REAR_VIEW:; N.R.S.

RESET_JOINT_DATA:; N.R.S

RESET_STOPWATCH:; N.R.S.

RESTORE_WIREFRAME_DISPLAY:; N.R.S.

RIGHT_SIDE_VIEW:; N.R.S.

SAVE_CELL: <cell>; N.R.S.

SET_DEVICE_MONITORED: <device>[,device] N.R.S.

SET_DISPLAY_INTERVAL: <real>; N.R.S.

SET_DISTANCE_UNITS: <units>; N.R.S.

SET_JOINT_DISPLAY: <PERCENTS_ONLY| N.R.S.
PERCENTS_AND_VALUES|WORST_JOINT_ONLY>
{,<PORTS|NOPORTS>};

SET_RECEIVE_MODE: {<device>},<port>,<MANUAL|AUTOMATIC>;N.R.S.

SET_SEND_MODE: {<device>},<port>,<MANUAL|AUTOMATIC>; N.R.S.

SET_SIMULATION_INTERVAL: <real>; N.R.S.

SET_TIMING_FILE: <timingfile>; N.R.S.

SET_TIMING_FILE_OFF:; N.R.S.

SET_TPOINT_DISPLAY_SIZE: <units>,<real>; N.R.S.

SET_TRACE_ON: <frameortpt>,<frame>,<frame>; N.R.S.

SET_TRACE_OFF: <frame>,{part}; N.R.S.

SET_VARIABLES: <variable>,<framename>,<tpointname>; N.R.S.

SLOWER:; N.R.S.

SMOOTH_SHADING:; N.R.S

STOPWATCH_OFF:; N.R.S.

STOPWATCH_ON:; N.R.S.

TPOINT_TO_FRAME: <tpoint>,<frame>; N.R.S.

TPOINT_TO_TPOINT: <tpoint>,<tpoint>; N.R.S.

TRACKING_OFF:; N.R.S.

TRACKING_ON:; N.R.S.

TRIMETRIV_VIEW:; N.R.S.

UNBLANK_FRAMES: <frame>[,frame]; N.R.S.

UNBLANK_PARTS: <frame>[,frame]; N.R.S.

UNBLANK_SOLIDS: N.R.S.
<SPHERES|BOXES|HULLS|ALL_SOLIDS>;

UNBLANK_SUBTREE: N.R.S.
<FRAMES|TPOINTS|PARTS>,<frame>;

UNBLANK_TEXT: N.R.S.
<ALL|(STOPWATCH|JOINTS|SEQWINDOW|MOVETEXT)>;

UNBLANK_TPOINTS: <frame>[,frame]; N.R.S.

UNBLANK_UP_TO_WORLD: N.R.S.
<FRAMES|TPOINTS|PARTS>,<frame>;

VIEW_CENTER_TPOINT: <frame|tpoint>; N.R.S.

### D.7.0.0. Important SRCL-command used when initializing

(they will have to be generated by command when starting the postprocessing)

| | | |
|---|---|---|
| DEF HP | define main program | |
| | AD | adress |
| | VZ | branch |
| | UP | subprogram |
| | ZY | cycle |

### D.8.0.0. SRCL-commands not incorporated in this mapping

| | |
|---|---|
| BS | conveyer synchronization |
| GRF | gripper, will probably be set by RDC or at end of movement. |
| NOP | no operation |
| NPK | zero-offset correction |
| ORI | orientation (wrist-rotation during path or at end point.) |
| PAU | Output to periphery |
| RDL | remaining loops cancelled |
| TV | tranfer results of binairy operation |
| TXT | text |

# Appendix E: SRCL commands.

**E.0.0.0. INDEX**

| Command description: | English Command: | German Command: |
|---|---|---|

**E.1.0.0. Program Sequence Instructions**

| | | |
|---|---|---|
| Definition | DEF | DEF |
| Main Program | MP | HP |
| Address | AD | AD |
| Branch | BR | VZ |
| Subprogram | SP | UP |
| Cycle | CY | ZY |
| Sensor Function | SF | SF |
| Space Point Data | SD | RD |
| Variable Data | VD | VD |
| Technology Table | TET | TEC |
| Jump | JMP | SPG |
| Main Program | MP | HP |
| Address | AD | AD |
| Branch | BR | VZ |
| Subprogram | SP | UP |
| Cycle | CY | ZY |
| Sensor Function | SF | SF |
| On | ON | EIN |
| Off | OFF | AUS |
| Wait | WAI | WRT |
| Time | T | Z |
| Input | I | E |
| Conditional | CON | BAW |
| Greater | GR | GR |
| Less Than | LE | KL |
| Equal | EQ | GL |
| Not Equal To | NE | UG |
| True | T | W |
| False | F | F |
| Halt | HLT | HLT |
| Unconditional | UN | UN |
| Conditional | CO | BE |
| Remaining Loops Cancel | RLC | RDL |
| Interrupt | INT | UNT |
| On | ON | EIN |
| Off | OFF | AUS |
| Execute | EX | BEA |
| Not Execute | NX | NBE |

## E.2.0.0. Movement Instructions

| | | | | |
|---|---|---|---|---|
| Linear Movement | LIN | | LIN | |
| Positions | | XYZ | | XYZ |
| Orientations | | ABC | | ABC |
| Auxiliary Axes | AAX | | ZAC | |
| On | | ON | | EIN |
| Off | | OFF | | AUS |
| Linear Aux Axis Move | LIN | | LIN | |
| Positions | | XYZ | | XYZ |
| Orientations | | ABC | | ABC |
| Aux Axes | | A1 - A6 | | Z1 - Z6 |
| Circular Movement | CR | | ZR | |
| Positions Point 1 | | XYZ | | XYZ |
| Positions Point 2 | | XYZ | | XYZ |
| Orientations | | ABC | | ABC |
| Circular Aux Axes Move | CIA | | ZRZ | |
| Positions Point 1 | | XYZ | | XYZ |
| Positions Point 2 | | XYZ | | XYZ |
| Orientations | | ABC | | ABC |
| Aux Axes | | A1 - A6 | | Z1 - Z6 |
| Point To Point Movement | PTP | | PTP | |
| Positions | | XYZ | | XYZ |
| Orientations | | ABC | | ABC |
| Point To Point Aux Axes | PPA | | PPZ | |
| Positions | | XYZ | | XYZ |
| Orientations | | ABC | | ABC |
| Aux Axes | | A1 - A6 | | Z1 - Z6 |
| Velocity | VEL | | GES | |
| Linear | | CPA | | BAN |
| All Axes | | ALL | | ALL |
| Specific Axis | | AXS | | ACH |
| Override | | OV | | OV |
| One Additional Axis | | AXX | | ZAC |
| All Additional Axes | | AAA | | ZAL |
| Acceleration | ACC | | BES | |
| Linear Motion | | CPA | | BAN |
| All Axes | | ALL | | ALL |
| Specific Axis | | AXS | | ACH |

| | | |
|---|---|---|
| Orientation | ORI | ORI |
| Fixed | FIX | KON |
| Variable | VAR | VAR |
| Approximate Positioning | APO | UES |
| Linear | CPA | BAN |
| Point To Point | PTP | PTP |
| Zero Offset | ZOF | NPK |
| Tool Offset | TOF | WZK |
| Position | T L D | T L D |

### E.3.0.0. Binary Logic Instructions

| | | | | |
|---|---|---|---|---|
| And | | A | | U |
| | Marker | | M | M |
| | Not Marker | | NM | NM |
| | Input | | I | E |
| | Not Input | | NI | NE |
| | Bit Store | | B | B |
| | Not Bit Store | | NB | NB |
| Or | | O | | O |
| | Marker | | M | M |
| | Not Marker | | NM | NM |
| | Input | | I | E |
| | Not Input | | NI | NE |
| | Bit Store | | B | B |
| | Not Bit Store | | NB | NB |

## E.4.0.0. Input/Output Memory Instructions

| | | |
|---|---|---|
| Transfer Results | TR | TV |
| Output | O | A |
| Marker | M | M |
| Bit Store | B | B |
| Set | S | S |
| Output | O | A |
| Marker | M | M |
| Impulse Output | IO | IA |
| Bit Store | B | B |
| Reset | RS | RS |
| Output | O | A |
| Marker | M | M |
| Bit Store | B | B |
| Load | LAD | LAD |
| Parameter | P | P |
| Output | O | A |
| Variable Memory | VAM | VSP |
| Variable | V | V |
| Space Point | S | R |
| Gripper | GRP | GRF |
| Current Position Store | POS | POS |
| Open | OPN | AUF |
| Close | CL | ZU |
| Output to Periphery | OUT | PAU |
| P-Word | PW | PW |
| M-Word | MW | MW |
| H-Word | HW | HW |

### E.5.0.0. Arithmetic Instructions

| Arithmetic | ARI | ARI |
|---|---|---|
| Addition | ADD | ADD |
| Substraction | SUB | SUB |
| Multiplication | MLT | MLT |
| Division | DIV | DIV |
| Compare | CMP | VGL |

## E.6.0.0. Special Instructions

| | | |
|---|---|---|
| Analog Output | ANO | ANA |
| Offset | O | O |
| Velocity | VEL | GES |
| Analog Table | AT | AT |
| Integration Time | ITE | IZT |
| Technology Table | TET | TEC |
| On | ON | EIN |
| Off | OFF | AUS |
| Conveyor Synchronization | CV | BS |
| Switch On | SON | ANF |
| Switch Off | OFF | END |
| Interrupt | INT | UNT |
| Time Distance Function | TDF | TDF |
| On | ON | ON |
| Off | OFF | OFF |
| Pendulum Motion | PND | PND |
| Axis 6 | AXS | ASE |
| Cartesian | CAR | KAR |

## E.7.0.0. Sensor Function Instructions

| | | |
|---|---|---|
| Sensor Analog Interface | SAI | SAS |
| On | ON | EIN |
| Off | OFF | AUS |
| Sensor Control | SCI | SST |
| Sensor Read | SRI | SDL |
| Once | ONC | EML |
| On | ON | EIN |
| Off | OFF | AUS |
| Sensor Data Request | SDR | SDA |
| Once | ONC | EML |
| Cyclic | CYC | ZYK |
| Cyclic Off | OFF | AUS |
| Load Sensor List | SLT | SLT |
| Variable Memory | VAM | VSP |
| Control Variable | CVA | FKN |
| Sensor Compare | SCM | SVG |
| Greater | GR | GR |
| Less | LE | KL |
| Equal | EQ | GL |
| Not Equal | NE | NG |
| Sensor Arithmetic | SAR | SAR |
| Addition | ADD | ADD |
| Substraction | SUB | SUB |
| Multiplication | MUL | MUL |
| Division | DIV | DIV |
| Square Root | SQR | WRZ |
| Sine | SIN | SIN |
| Cosine | COS | COS |
| Sensor Binary Logic | SBO | VKM |
| And | A | U |
| Or | O | O |
| Exclusive Or | XOR | XOR |
| Not | N | N |
| Sensor Load | SLA | SLA |
| Variable Memory | VAM | VSP |
| Variable | V | V |
| Space Point | S | R |

| | | | | |
|---|---|---|---|---|
| Comparator | CCM | | KMP | |
| Input Memory | | INM | | ESP |
| Variable Memory | | VAM | | VSP |
| Binary Velocity Control | BVC | | GFB | |
| Bit Memory | | B | | B |
| Input | | I | | E |
| Binary Path Control | BCP | | BKB | |
| Bit Memory | | B | | B |
| Input | | I | | E |
| Analog Path Control | ACP | | BKA | |
| Input Memory | | INM | | ESP |
| Variable Memory | | VAM | | VSP |
| Analog Velocity Control | AVC | | GFA | |
| Input Memory | | INM | | ESP |
| Variable Memory | | VAM | | VSP |

### E.8.0.0. Text display Instructions

| | | |
|---|---|---|
| No Operation | NOP | NOP |
| Remark | REM | KOM |
| Text | TXT | TXT |