# User Programming

## Introduction

SIMION 7.0 incorporates a very powerful feature called User Programs. **This feature allows you to model ion traps, quadrupoles, RF tuned devices, time-of-flight components, collision cells, and all sorts of other things.** As you become familiar with user programming, it will become apparent that you are really only limited by your imagination.

Whenever SIMION loads either a **.PA** or **.PA0** file it looks for an associated **.PRG** user program file. All such user program files will be automatically compiled and used when flying ions to modify SIMION's behavior (*ONLY when an ion is in an instance that uses a potential array that has associated user programs e.g. a* ***.PRG*** *file*). These program fragments run about 2-5 times slower than direct C code modifications of SIMION (*quite fast*).

SIMION contains a User Program debugger/compiler (*accessed via the* **Test & Debug** *button on the Main Menu Screen*) to assist you in the testing, debugging, and development of your user programs. The EDY editing program supplied with SIMION (*or an editor of your choice - see EDY - Appendix H*) can be accessed from within the debugger to create/modify these user programs.

## What Is a User Program ?

A user program is an ASCII file that contains one or more program segments (*e.g. sub-programs*) written in an HP RPN calculator *style* language. A user program file (*and its program segments*) is always associated with ***one and only one*** potential array. As an ion flies within an array instance that projects a potential array with associated user program segments, SIMION ***automatically*** calls each program segment at the appropriate times to allow it to control how the ion flies.

> ***Trick: Use a crude no-field 3D array (with user programs) sized to workspace volume as instance 1 to control ions outside normal instances.***

These program segments can dynamically change fast adjust and/or fast scale electrodes; electrostatic and magnetic fields; ion accelerations; and all sorts of other things. A user program file has the same name as the potential array name it supports and the extension **.PRG** (*e.g.* **TEST.PRG** *is the user program file for* **TEST.PA**). *User Programs can be turned off (See Adj Var. Section).*

## Program Segments Within a User Program File

A user program file is composed of an ***optional*** **Define_Data** segment ***and*** from ***one*** to ***nine*** program segments. Each included program segment has a different specific purpose and particular access and control capabilities. SIMION's user program feature is designed to allow any legal combination of user program segments to work together to provide the desired modeling. User programs are much like subroutines because they can communicate, share data, and otherwise support each other. The material below gives a very brief introduction to these program segments:

### Seg Define_Data

This segment is ***always the first segment*** in any user program file (*you are not required to declare it - assumed first segment by default*). It contains the definitions for global variables and array variables (*visible to **all** active user program segments - in **all** instances that support user programs*). These variables are used for storage and communication between user programs.

Two types of global variables and array variables are supported: **Adjustable** and **Static**. **Adjustable** variables are displayed/adjusted at the beginning of each **Fly'm** (*adjustable variables can also be **user accessed while** ions are flying*) to allow you to change or adjust the function of your user program segments without having to edit them. The lifetime of Adjustable variables is that of the total fly'm including any reruns. **Static** variables are not user adjustable but are directly accessible to all active user programs for other control and storage functions. The lifetime of Static variables is the lifetime of the current ion or group of ions ***while*** they are flying.

### The Nine Types of User Program Segments

The nine types of program segments utilize a powerful monitor, analyze, and modify paradigm. Each type of user program segment is **only** allowed to perform certain specific functions. In general, SIMION calculates something ***first*** (*e.g. the next time step to use*) and ***then*** calls a specific program segment (*e.g. the program segment Tstep_Adjust*) ***if*** it is defined in the ion's current instance. The called user program segment can then monitor, analyze, and modify its allowed parameters as needed (*e.g. the next time step to use*). Many types of user programming segments may be able to monitor a parameter (*e.g. ion velocity*) but only a few are allowed to modify it (*e.g. ion velocity – Initialize and Other_Actions*). This provides real power without permitting total chaos.

***Only one type of each program segment is allowed in any one user program file.*** The following is a brief introduction to each program segment type:

#### Program_Segment Initialize

The **Initialize** segment is used to dynamically change an ion's initial parameters and conditions. This program segment can output messages and control looping back for another run (*e.g. useful for automatic focusing user programs*).

#### Program_Segment Init_P_Values

A program segment called **Init_P_Values** initializes (*if it exists*) via fast adjust and/or fast scaling methods entire potential arrays **before** flying **any** ions. **Note**: Unlike ***all*** the other program segments, ions do not ***have*** to be in the instance to have the **Init_P_Values** program segment called. ***This also means that the ion and instance context have no meaning within this program segment (e.g. ion and instance related variables are not accessible)***.

#### Program_Segment Tstep_Adjust

The **Tstep_Adjust** segment can be used to examine and possibly change the integration time step (*in microseconds*). This program segment (*if it exists*) is called ***after*** SIMION has determined the next integration time step and just ***before*** the integration time step is performed.

#### Program_Segment Fast_Adjust

The **Fast_Adjust** segment can be used to examine/adjust the electrode/pole potentials of fast adjustable ***and/or*** fast scaleable array instances (e.g. *with* **.PA0** *potential arrays*) as the ion flies. This program segment (*if it exists*) is called to adjust the array's potentials ***before*** any initial field determinations are made.

### Program_Segment Efield_Adjust

The **Efield_Adjust** segment is used to examine/change the electrostatic field potentials and gradients calculated for each time step. ***Note: this segment type can only be used with electrostatic potential arrays.***

### Program_Segment Mfield_Adjust

The **Mfield_Adjust** segment is used to examine/change the magnetic fields calculated for each time step. ***Note: this segment type can only be used with magnetic potential arrays.***

### Program_Segment Accel_Adjust

The **Accel_Adjust** segment is used to examine/change the acceleration components for each time step.

### Program_Segment Other_Actions

The **Other_Actions** segment is called just after each time step. It is used to examine/change ion parameters like: Mass, velocity, splat, and etc. Moreover, the user can output messages and results to the data recording screen and file.

### Program_Segment Terminate

The **Terminate** segment is called just after ***all ions*** have died (*splat*). It is used to examine ion parameters like: Mass, velocity, and etc. This program segment can output messages and control looping back for another run (*e.g. useful for automatic focusing user programs*).

## Two Examples of a SIMION User Program File

The following listings are of a simple **Accel_Adjust** program segment for adding Stokes Law viscosity effects that were created with two very different programming styles:

### A Questionable Programming Style

```
defa viscous_damping 0,seg accel_adjust rcl ion_ax_mm
rcl ion_vx_mm rcl viscous_damping * - sto ion_ax_mm
rcl ion_ay_mm rcl ion_vy_mm rcl viscous_damping * -
sto ion_ay_mm rcl ion_az_mm rcl ion_vz_mm
rcl viscous_damping * - sto ion_az_mm
```

### A Suggested Programming Style

```
; This Accel_Adjust program segment adds a simple Stokes' Law Viscosity Effect.
; It serves as a starting point for dabbling in ion mobility and atmospheric ion sources.
; The program segment makes use of a simple viscous damping factor and linear viscosity.
; Note:  This program segment has problems with high viscosity (see page I-29 for fixed version).

;Note:  a Begin_Segment Define_Data is not required (the compiler assumes it)

        Define_Adjustable Viscous_Damping 0     ; adjustable variable Viscous_Damping
                                                ; set to 0 (no viscous damping by default)
                                                ; adjustable at the beginning of each Fly'm
```

```
Begin_Segment Accel_Adjust                ; start of Accel_Adjust program segment

        Recall Ion_Ax_mm                  ; recall current x acceleration (mm/usec²)
        Recall Ion_Vx_mm                  ; recall current x velocity (mm/sec)
        Recall Viscous_Damping            ; recall the viscous damping term
        Multiply                          ; multiply times x velocity
        Subtract                          ; and subtract from x acceleration
        Store Ion_Ax_mm                   ; return adjusted value to SIMION

        Recall Ion_Ay_mm                  ; recall current y acceleration (mm/usec²)
        Recall Ion_Vy_mm                  ; recall current y velocity (mm/sec)
        Recall Viscous_Damping            ; recall the viscous damping term
        Multiply                          ; multiply times y velocity
        Subtract                          ; and subtract from y acceleration
        Store Ion_Ay_mm                   ; return adjusted value to SIMION

        Recall Ion_Az_mm                  ; recall current z acceleration (mm/usec²)
        Recall Ion_Vz_mm                  ; recall current z velocity (mm/sec)
        Recall Viscous_Damping            ; recall the viscous damping term
        Multiply                          ; multiply times z velocity
        Subtract                          ; and subtract from z acceleration
        Store Ion_Az_mm                   ; return adjusted value to SIMION

Exit                                      ; exit to SIMION (optional statement)
```

## *Language Rules and Machine Model*

***Both of the above programs will generate the same compiled code and will run at the same speed.*** However, the second example will be easy to support and modify later. Remember, ***you*** have the freedom to make your programs as cryptic or verbose as you like. There are several characteristics of the language that should be apparent:

### Upper and Lower Case

SIMION ***ignores the case*** of the statements entered. ***You may use upper and lower case freely to improve readability.***

### Blank Lines and Indention's

Blank lines are ignored. Use blank lines to create good visual separation of various regions of a program. You may indent code as desired. When properly used, indention's can significantly improve code readability.

### The Semicolon ; Starts an In-line Comment

In-line comments begin with a semicolon (*use a leading space or comma in front of the semicolon for separation from any preceding word*). All information after the semicolon (*including the semicolon*) is ignored by the compiler. **Unlike interpreted programs, these comments have no effect on the speed of user program execution (so use them!).**

## Word Oriented Language Structure

The user programming language makes use of a word oriented structure. This means that a program is a collection of words ***separated by any number of spaces, commas, tabs, and/or lines***. These words are analyzed to create the pseudo-machine code that SIMION actually executes. Lines have no meaning except that **all words beyond column 200 will be ignored**.

SIMION checks each word to see if it is a command, number, reserved variable, variable, or label (***in that order***).

## Command Words

SIMION's language treats all command words (*e.g.* **STO**) as reserved words (**cannot be used for any other purpose - except in comments**). Note: Most commands have synonyms (*e.g.* **STO** *and* **STORE**). This allows you more freedom of expression in making your programs as cryptic or verbose as desired. Each command will be covered in detail below.

## Numbers

***Any word that can be converted into a number will be!*** Thus commands or names cannot be numbers (*e.g. 1.24e-5 is a number, 12345t is not a number*). A number appearing where the compiler expects a command will be interpreted as **Recall Constant** (*of the value of the number*).

## The RPN Registers

SIMION makes use of a ten register rotary stack. Movement around the stack is automatic via the insertion and combining of numbers (*double precision - 64 bit*). The stack ***pointer*** rolls around the stack so that it always points at the ***current number*** (*last entered or a command result*).

The ***current number*** is always designed in the x-register. The number directly above it (*preceding it*) is in the y-register, above it is the z-register, then the t-register, and so on.

- Functions like **SIN** replace the current value of the x-register with its sine (*the x-register pointer is unchanged*).
- Other functions like + add the x and y register values together and place the result in the y-register which now automatically ***becomes*** the new x-register (*the x-register pointer is rolled up one register*).
- Entering a number ***or*** recalling a variable places the new number in the register directly ***below*** the current x-register. This register now automatically ***becomes*** the new x-register (*the x-register pointer is rolled down one register*).

## Variable Names and Labels

Words that aren't commands and can't be converted into numbers are considered to be candidates for variable names and labels. A variable name or label follows C naming conventions. The first character must be a letter or underscore (*e.g.* _ ). All remaining characters must be letters, numbers, or underscores. The first **31** characters of variable names and labels are significant (*for matching purposes*). Unlike C, SIMION ***ignores the case*** of the variable names and labels (*it retains case for display purposes only*).

Moreover, certain names are used for **reserved variables**. These variables (*e.g.* **Ion_Time_of_Flight**) allow you to exchange information with and exert control over SIMION. Each of the **reserved variables** will be covered in detail below.

## Unit, Orientation, and Angular Conventions

### SIMION's Standard Unit Systems

The basic position/length unit is millimeters (*mm*) or grid units (*gu*) depending on the command or reserved variables. Time is measured in microseconds (*µsec*). Velocity is mm/µsec. Acceleration is in mm/µsec$^2$. Magnetic fields are in Gauss. Electrostatic gradients are in volts/mm or volts/gu (*depending on the reserved variable*).

Three unit systems are used in connecting user programs with SIMION (*via Reserved variables*):

1. The first unit system is the **currently aligned workbench coordinates** (*mm*) and orientation (*Variables using these coordinates/orientations have names ending with _mm*). Variables using this unit system share the locations and orientations of the ***currently aligned workbench coordinates*** (*including* **Align** *button status*).
2. The second unit system is the ion's current instance's **PA volume coordinates** (*gu*) and orientation (*Variables using these coordinates/orientations have names ending with _gu*). **PA volume coordinates** are the 3D instance coordinates (*in gu*) displayed by **where** in View. Reversible 3D transformations can be performed between **currently aligned workbench coordinates** and **PA volume coordinates** of the ion's current instance.
3. The third unit system is the ion's current instance's PA Array coordinates ( *Variables using these coordinates have names ending with _Abs_gu*). ***Note:*** Coordinate transformations from 3D PA volume coordinates into the 2D or 3D coordinates of the actual potential array are ***non-reversible***. For 3D arrays x, y, and z are converted into their absolute values. For 2D planar x and y are converted to their absolute values and z is set to zero. For 2D cylindrical x is converted to its absolute value, y and z are converted into r, which is stored in y, and z is set to zero.

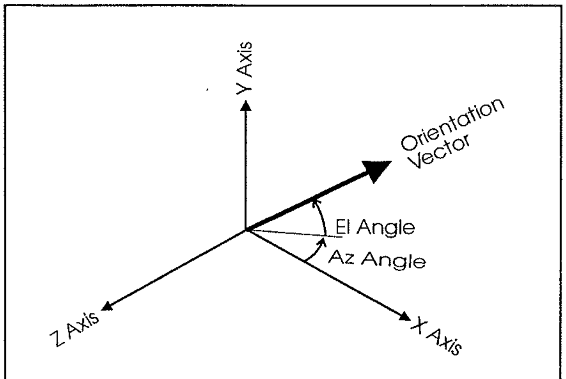
### SIMION's Angular Conventions

Several user program commands make use of angular input and output parameters (*e.g.* **az** *and* **el**). Angular parameters are either in ***degrees*** or ***radians***. Each command that makes use of angles will state whether the angles are in ***degrees*** or ***radians***.

Commands making use of azimuth and/or elevation angles follow the following conventions:

**az** Azimuth angle to apply when projecting the internal coordinates into external coordinates. Azimuth angle is ***degrees*** of ccw rotation about the y-axis in degrees looking down the positive y-axis toward the origin.

***Azimuth of 90 degrees.*** Internal z-axis made parallel to external x-axis. Internal x-axis made parallel to external negative z-axis. Internal y-axis remains parallel to



Angular Orientations

external y-axis (*assuming el = 0*).

**el** Elevation angle to apply when projecting the internal coordinates into external coordinates. Elevation angle is ***degrees*** of ccw rotation about the z-axis in degrees looking down the positive z-axis toward the origin .

***Elevation of 90 degrees***. Internal x-axis made parallel to external y-axis. Internal y-axis made parallel to external negative x-axis. Internal z-axis remains parallel to external z-axis (*assuming az = 0*).

Azimuth and elevation transformations are applied in the following order:

1. The elevation (*el*) transformation is applied creating an interim coordinate system.
2. The azimuth (*az*) transformation is then applied to the interim coordinate system to create the resulting coordinate system.

## PROGRAMMING COMMANDS

The following is a detailed discussion of each legal user programming command including examples of use. ***Unless stated otherwise, each command is legal in any program segment***. Command synonyms (*if any*) appear after the ***or:***

### + or: Add

Adds contents of x and y registers, puts result in y-register, and renames it as x-register ( *e.g.* **1 2 +** *becomes 3 in register where 1 was originally stored*).

### - or: Subtract

Subtracts x from y, puts result in y, and renames it x (*e.g.* **11 5 -** *becomes 6 in register where 11 was originally stored*).

### * or: Multiply

Multiplies x and y, puts result in y, and renames it x (*e.g.* **5 6 *** *becomes 30 in register where 5 was originally stored*).

### / or: Divide

Divides x into y, puts result in y, and renames it x (*e.g.* **60 10 /** *becomes 6 in register where 60 was originally stored*).

### 1/X or: Reciprocal_of_X

Converts the contents of the x-register to its reciprocal (*e.g.* **10.0 1/X** *becomes 0.1*).

### 10^X or: 10_ to_the_X

Converts the contents of the x-register to $10^x$ (*e.g.* **3 10^X** *becomes 1000*).

## >ARR or: PA_Coords_to_Array_Coords

Converts 3D point from the current instance's **P**otential Array **volume** coordinates to **actual** potential **Array** coordinates *(position according to array type: 2D, Cylindrical, 3D and etc.)*. On entry the x, y, and z registers are assumed to contain the point's x, y, and z **PA volume** coordinates (*in gu – grid units*). On exit the x, y, and z registers contain the point's actual **Array** coordinates (*in gu*). ***This is not a reversible transformation***. For 3D arrays x, y, and z are converted into their absolute values. For 2D planar x and y are converted to their absolute values and z is set to zero. For 2D cylindrical x is converted to its absolute value, y and z are converted into r, which is stored in y, and z is set to zero.

## >AZR or: Azimuth_Rotate

Rotates a 3D vector in the azimuth direction (***rotation around the y component axis***. *e.g. for -90 degrees old x component becomes new z component*). On entry the y, z, and t registers are assumed to contain the vector's x, y, and z components respectively. The x-register contains the azimuth angle of rotation (*in **degrees*** **ccw** *from x-axis looking down the positive y axis toward the origin*). On exit the x, y, and z registers contain the vector's rotated x, y, and z components.

## >DEG or: Radians_to_Degrees

Converts value in x-register from assumed ***radians to degrees***(*e.g.* **3.1459 >DEG** *becomes 180*). The **>RAD** command performs the reverse transformation.

## >ELR or: Elevation_Rotate

Rotates a 3D vector in the elevation direction (***rotation around the z component axis***. *e.g. for +90 degrees old x component becomes new y component*). On entry the y, z, and t registers are assumed to contain the vector's x, y, and z components. The x-register contains the elevation angle of rotation (*in degrees* **ccw** *from x-axis looking down the positive z axis toward the origin*). On exit the x, y, and z registers contain the vector's rotated x, y, and z components.

## >KE or: Speed_to_Kinetic_Energy

Converts from speed (*mm/µsec*) to kinetic energy (*eV*). On entry the x-register is assumed to contain the ion's speed (*mm/µsec*) and the y-register is assumed to contain the mass of the ion (*amu*). On exit the x-register contains the ion's KE (*eV*) and the y-register is unchanged. ***This transform uses relativistic corrections***. The **>SPD** command performs the reverse transformation.

## >P or: Rect_to_Polar

Converts from 2D rectangular to 2D polar coordinates. On entry the x-register is assumed to contain the x value and the y-register is assumed to contain the y value. On exit the x-register contains the radius and the y-register is contains the angle theta ***in degrees***. The **>R** command performs the reverse transformation.

## >P3D or: Rect3D_to_Polar3D

Converts from rectangular 3D to polar 3D coordinates. On entry the x, y, and z register are assumed to contain the rectangular x, y, and z vector components. On exit the x-register contains r (*radius*), the y-register contains the azimuth angle ***in degrees***, and the z-register the elevation angle ***in degrees***. The **>R3D** command performs the reverse transformation.

### >PAC  or: WB_Coords_to_PA_Coords

Converts 3D point from workbench coordinates (*in **mm** - current workbench alignment*) to **P**otential **A**rray volume coordinates ( *in **gu** - from ion's current instance's working origin*). On entry the x, y, and z registers are assumed to contain the point's x, y, and z **WB** coordinates (*mm*). On exit the x, y, and z registers contain the point's **PA** volume coordinates (*gu*). The **>WBC** command performs the reverse transformation.

### >PAO  or: WB_Orient_to_PA_Orient

Converts 3D vector from workbench orientation (*current alignment*) to **P**otential **A**rray volume orientation (*of ion's current instance*). On entry the x, y, and z registers are assumed to contain the vector's x, y, and z **WB** components. On exit the x, y, and z registers contain the vector's **PA** components. ***Note: The magnitude of the vector is not changed. Its three component vectors are now aligned with the current instance's x, y, and z axis.*** The **>WBO** command performs the reverse transformation.

### >R  or: Polar_to_Rect

Converts from 2D polar to 2D rectangular coordinates. On entry the x-register is assumed to contain r (*radius*) and the y-register is assumed to contain the angle theta ***in degrees***. On exit the x-register contains the x value and the y-register contains the y value. The **>P** command performs the reverse transformation.

### >R3D  or: Polar3D_to_Rect3D

Converts from polar 3D to rectangular 3D coordinates. On entry the x-register is assumed to contain r (*radius*), the y-register is assumed to contain the azimuth angle ***in degrees***, and the z-register the elevation angle ***in degrees***. On exit the x, y, and z registers contain the rectangular x, y, and z vector components. The **>P3D** command performs the reverse transformation.

### >RAD  or: Degrees_to_Radians

Converts value in x-register from assumed ***degrees to radians***(*e.g.* **180 >RAD** *becomes 3.1459*). The ***>DEG*** command performs the reverse transformation.

### >SPD  or: Kinetic_Energy_to_Speed

Converts from kinetic energy (*eV*) to ion speed (*mm/μsec*). On entry the x-register is assumed to contain the ion's KE (*eV*) and the y-register is assumed to contain the mass of the ion (*amu*). On exit the x-register contains the ion's speed (*mm/μsec*) and the y-register is unchanged. ***This transform uses relativistic corrections***. The **>KE** command performs the reverse transformation.

### >WBC  or: PA_Coords_to_WB_Coords

Converts 3D point from the current instance's **P**otential **A**rray volume coordinates (*in **gu** – ion's current instance's working origin*) to work**b**ench coordinates (*in **mm** – current workbench alignment*). On entry the x, y, and z registers are assumed to contain the point's **PA** x, y, and z volume coordinates (*in **gu***). On exit the x, y, and z registers contain the point's **WB** Coordinates (*in **mm***). The **>PAC** command performs the reverse transformation.

**>WBO** **or: PA_Orient_to_WB_Orient**

Converts 3D vector from the ion's current instance's **P**otential **A**rray volume orientation to **w**ork**b**ench orientation (*current alignment*). On entry the x, y, and z registers are assumed to contain the vector's **PA** x, y, and z aligned components. On exit the x, y, and z registers contain the vector's **WB** aligned components. ***Note: The magnitude of the vector is not changed, only orientation of its vector components***. The **>PAO** command performs the reverse transformation.

**ABS** **or: Absolute_Value**

Converts the contents of the x-register to a positive number (*e.g.* **-2.5 ABS** *becomes 2.5*).

**ACOS** **or: Arc_Cosine**

Converts the contents of the x-register to arc cosine (***in radians***) (*e.g.* **1.0 ACOS** *becomes 0.0*).

**ADEFA** **or: Array_Define_Adjustable**
**ADEFA Name Size ; "filename"** **(e.g. ADEFA Energy 100 ; "energy.dat")**
**Only Legal in Define_Data Segment**

Three word command that defines an ***adjustable array*** named ***Name*** with a size of ***Size*** elements (*must be 1 or greater*). ***Name*** must not conflict with any reserved word or previously defined variable (*any type*) or label. Example: **ADEFA Energy 100 ; "energy.dat"** Means define an adjustable array named Energy, 100 elements in size. ***Pre-zero*** and ***then*** auto-initialize the array at the ***beginning*** of each **Fly'm** (*but* ***not*** *each rerun*) with the file **energy.dat** (*see page I-19 for initialization file format*).

*Note: The initialization file is optional*. If provided, it must appear somewhere within the following inline comment (*after a ';'*) and be enclosed in quotes (""). If there are more values in the optional initialization file than the defined array size, SIMION will only read the number of values required to fill the array. If there are fewer values in the file than the defined array size, SIMION will load the values provided, starting with the first array element, and the remaining array elements will remain zeroed.

**ADEFS** **or: Array_Define_Static**
**ADEFS Name Size ; "filename"** **(e.g. ADEFS Voltage 400 ; "voltages.dat")**
**Only Legal in Define_Data Segment**

Three word command that defines a ***static array*** named ***Name*** with a size of ***Size*** elements (*must be 1 or greater*). ***Name*** must not conflict with any reserved word or previously defined variable (*any type*) or label. Example: **ADEFS Voltage 400 ; "voltages.dat"** Means define a static array named Voltage, 400 elements in size. ***Pre-zero*** and ***then*** auto-initialize the array ***just before*** flying each ion (*or group*) with the file **voltages.dat** (*see page I-19 for initialization file format*).

*Note: The initialization file is optional*. If provided, it must appear somewhere within the following inline comment (*after a ';'*) and be enclosed in quotes (""). If there are more values in the optional initialization file than the defined array size, SIMION will only read the number of values required to fill the array. If there are fewer values in the file than the defined array size, SIMION will load the values provided, starting with the first array element, and the remaining array elements will remain zeroed.

**ALOAD** **or: Array_Load**
**ALOAD Name ;"filename" (e.g. ALOAD Energy ; "energy.dat")**
**Adjustable Arrays: Legal in any Program Segment**
**Static Arrays: Illegal in Initiate and Terminate Segments**

Three word command that loads an ***array*** named ***Name*** with the contents of the file ***filename***. ***Name*** must be a previously defined adjustable or static array. **ALOAD** will read the same free format ASCII file format as described above for initialization files. Example: **ALOAD Energy ; "energy.dat"** Means ***pre-zero*** ***all*** elements of the array named Energy and ***then*** load them with the contents of the **energy.dat** file (*see page I-19 for initialization file format*).

*Note: The filename is required*. It must appear somewhere within the following inline comment (*after a ';'*) and be enclosed in quotes (""). If there are more values in the file than the defined array size, SIMION will only read the number of values required to fill the array. If there are fewer values in the file than the defined array size, SIMION will load the values provided starting with the first array element, and the remaining array elements will remain zeroed.

**ARCL** **or: Array_Recall**
**ARCL Name**
**Adjustable Arrays: Legal in any Program Segment**
**Static Arrays: Illegal in Initiate and Terminate Segments**

Uses the current value of the x-register as a index value to array ***Name***, and replaces the index value in the x-register with the value of the array element designated by the index value. The stack pointer remains unchanged. If the index value is beyond the array limits (*<1 or >size*) a runtime error help screen will be generated. (*e.g.* **30 ARCL VOLTS** *recalls the value of the* $30^{th}$ *element of the array* **VOLTS** *and inserts it into the x-register replacing its previous value of 30*).

This command is designed to minimize stack clutter by replacing the specified array index with the array element's value. The following 2D array index computation serves to demonstrate:

```
RCL Y  RCL NX * RCL X +     ;index = x + nx * y   2D array mapped into 1D array
ARCL  ARRAY_2D              ;x-register has value - no index clutter remains on stack
```

**ASAVE** **or: Array_Save**
**ASAVE Name ;"filename" (e.g. ASAVE Energy ; "energy.dat")**
**Adjustable Arrays: Legal in any Program Segment**
**Static Arrays: Illegal in Initiate and Terminate Segments**

Three word command saves the values of ***all*** elements of an ***array*** named ***Name*** to the file ***filename***. ***Name*** must be a previously defined adjustable or static array. If the file ***filename*** already exists, its previous contents will be ***destroyed***. The format of the saved file is ASCII with five comma separated numbers per line. Example: **ASAVE Energy ; "energy.dat"** Means save all elements of the array named Energy to the energy.dat file.

*Note: The filename is required*. It must appear somewhere within the following inline comment (*after a ';'*) and be enclosed in quotes ("").

**ASIN** **or: Arc_Sine**

Converts the contents of the x-register to arc sine (***in radians***) (*e.g.* **1.0 ASIN** *becomes 1.570796*).

**ASTO** **or: Array_Store**
**ASTO Name**
**Adjustable Arrays: Legal in any Program Segment**
**Static Arrays: Illegal in Initiate and Terminate Segments**

Stores the value in the y-register to the array element in the array ***Name*** designated by the index value in x-register. The stack pointer is ***rolled up*** by one (*e.g. the old y-register is now the new x-register*). If the index value is beyond the array limits (*<1 or >size*) a runtime error help screen will be generated. (*e.g.* **RCL V1 15 ASTO VOLTS** *recalls the value of the variable* **V1** *and stores this value in the* $15^{th}$ *element of the array* **VOLTS** *and then rolls up the register pointer by one. The x-register now points to the value loaded by* **RCL V1**).

This command is designed to minimize stack clutter by rolling up the stack pointer by one ***after*** command execution. The following 3D array index computation serves to demonstrate:

| | |
|---|---|
| **50 ARCL VOLTS** | ;load x-register with $50^{th}$ element of array volts (item to save) |
| **RCL Z RCL NY * RCL Y +** | |
| **RCL NX * RCL X +** | ;index = x + nx *( y + ny * z) 3D array mapped into 1D array |
| **ASTO ARRAY_3D** | ;x-register points to starting value - no index clutter remains |

**ATAN** **or: Arc_Tangent**

Converts the contents of the x-register to arc tangent (***in radians***) (*e.g.* **1.0 ATAN** *becomes 0.785398*).

**BEEP** **or: Beep_Sound**

Makes a beep sound (*e.g.* **BEEP** ---> *beep!!*). Same sound as BELL command.

**BELL** **or: Ring_Bell**

Rings computer's bell (*e.g.* **BELL** ---> *beep!!*).

**CHS** **or: Change_Sign**

Reverse the sign of the number in the x-register (*e.g.* **2.0 CHS** *becomes -2.0*).

**CLICK** **or: Click_Sound**

Makes a click sound (*e.g.* **CLICK** ---> *click!!*).

**COS** **or: Cosine**

Converts the contents of the x-register (***in radians***) to cosine (*e.g.* **0.0 COS** *becomes 1.0*).

**DEFA** **or: Define_Adjustable**
**DEFA Name Number** **(e.g. DEFA Omega 1.0)**
**Only Legal in Define_Data Segment**

Three word command defines an ***adjustable variable*** named ***Name*** with an initial value of ***Number*** (***must be an actual number***). ***Name*** must not conflict with any reserved word or previously defined variable (***any type***) or label. Example: **DEFA Omega 1.0** Means define adjustable variable named Omega with initial value of 1.0. ***Can only appear in a Define_Data segment***. See discussion on variables below for more information.

**DEFS** **or: Define_Static**
**DEFS Name Number** **(e.g. DEFS Flag 0.0)**
**Only Legal in Define_Data Segment**

Three word command defines a ***static variable*** named ***Name*** with an initial value of ***Number*** (***must be an actual number***). ***Name*** must not conflict with any reserved word or previously defined variable (***any type***) or label. Example: **DEFS Flag 0.0** Means define static variable named Flag with initial value of 0.0. ***Can only appear in a Define_Data segment***. See discussion on variables below for more information.

**E^X** **or: E_to_the_X**

Converts the contents of the x-register to $e^x$ (*e.g.* **1.0 E^X** *becomes 2.71828*).

**ENTR** **or: Enter Duplicate_X**

Rolls the register pointer ***down*** one and copies the contents of the old x-register into the new x-register (*e.g.* **3 ENTR** * *duplicates 3, squares it and places 9 in initial x-register*).

**EXIT**

Exits the current program segment and returns to SIMION (*e.g.* **3 X>0 EXIT ENTER** *exits program segment because 3 is greater than zero*). ***This is the default command loaded into all of program memory***.

**FRAC** **or: Decimal_Fraction**

Converts the contents of the x-register to its decimal fraction component (*e.g.* **2.58 FRAC** *becomes 0.58*).

**GSB** **or: GoSub Go_Subroutine**
**GSB Label**

Unconditional jump to subroutine at label named **Label**. **Label** name must be a legal label defined elsewhere in the ***same program segment*** by a **LBL** statement. The first encounter of a **RTN** command within the subroutine will cause execution to resume just after the calling **GSB Label** statement. **Subroutines can be nested up to 100 levels deep.** Example: **GSB Double** means jump to the label named **Double** (*a subroutine*) and return after encountering a **RTN** statement.

**GTO** **or: Goto Go_To**
**GTO Label**

Unconditional jump to label named **Label**. **Label** name must be a legal label defined elsewhere in the ***same program segment*** by a **LBL** statement. Example: **GTO Entrya** means jump to the label named **Entrya**.

**INT** **or: Integer**

Converts the contents of the x-register to its integer component (*e.g.* **2.58 INT** *becomes 2.0*).

**KEY?** **or: Check_For_Key_Input**
**Only Legal in Other_Actions**

Checks for keyboard input, and inserts the ***upper case*** key code in x-register **after rolling pointer down one register**. If no keyboard input is available a zero is placed in the x-register. **The actual key codes generated can be found with the KEY? test button in the debugger.**

**LBL** **or: Label Entry Subroutine**
**LBL Label**

Marks a code entry point (*jump or subroutine*) with the name of Label. Label name must not conflict with any reserved word or previously defined variable (*any type*) or label. Example: **LBL Double** means an entry point called **Double**. *Note: SIMION will only allow jumps or subroutine calls to locations within the same program segment*.

**LN** **or: Natural_Log**

Converts the contents of the x-register to natural logarithm (*e.g.* **2.71828 LN** *becomes 1.0*).

**LOG** **or: Base_10_Log**

Converts the contents of the x-register to $base_{10}$ logarithm (*e.g.* **1000 LOG** *becomes 3.0*).

**MARK** **or: Mark_All_Ions**
**Only Legal in Other_Actions Segment**

Sets markers for all ions. Useful for making visual marks and as an event to trigger data recording. ***Note: Forces drawing of current ion trajectory line segment.*** Handy for forcing clean changes in ion color and etc.

**MESS** **or: Message**
**MESS ; X reg = # and Y reg = #**
**Only Legal in Initialize, Other_Actions, & Terminate**

Displays the following (*same line*) comment (***50 chars max*** *- without the ;*) as a data record line. Useful for user prompts and recording data. **Note: Each # character is replaced by a register value (*the first left-to-right # is x register value the second is y register value and so on*).** In the example above (*assuming x = 23 and y =125.3*) the output would be: **X reg = 23 and Y reg = 125.3**. *The actual format used for displaying the # numbers is the same as currently defined for data recording*.

**NINT** **or: Nearest_Integer**

Replaces the contents of the x-register to its nearest integer value (e.g. **1.9 NINT** becomes 2.0). Useful to insure precise whole values when testing numbers for being equal. The equals test can often be uncertain for floating point numbers (*requires all bits to be the same to be equal*).

## NOP

The NOP command does nothing. It can be used to fill space or kill time (*e.g.* **NOP NOP NOP NOP KEY?** *kills a little time before a key test*).

## R/S — or: Run/Stop
**Only Legal in Initialize, Other_Actions & Terminate**

Program halts execution, informs user on the display, and requests a keystroke to resume. The ***upper case*** key code for the key entered will be in the new (*rolled down by one*) x-register when execution resumes. **The actual key codes generated can be found with the KEY? test button in the debugger.**

## RAND — or: Random_Number

Rolls the register pointer down one and inserts a random number between 0 and 1 into the new x-register. SIMION 7.0 uses a new pseudo-random number generator. See Appendix E (*E-15*) for more details.

## RCL — or: Recall
## RCL Name

Rolls the register pointer down one and inserts the value of the variable **Name** in the new x-register. **Name** must be a currently active variable name that the user program segment is allowed to reference (*e.g.* ***RCL TEMP1*** *recalls value of variable named* ***TEMP1***). See more information below concerning: Adjustable, static, reserved, and temporary variables.

## REDRAW — or: Redraw_Screen
**Only Legal in Initialize, Other_Actions, & Terminate**

Redraws current View window. Useful if you want to erase unsaved trajectory vectors.

## RLDN — or: Roll_Register_Pointer_Down

Rolls the x-register pointer ***down*** by one (*or registers up by one* ***HP convention***). The current y-register was the prior x-register (*e.g.* **5 10 RLUP RLDN** *has 10 in the new x-register*).

## RLUP — or: Roll_Register_Pointer_Up

Rolls the x-register pointer ***up*** by one (*or registers down by one* ***HP convention***). The current x-register was the prior y-register (*e.g.* **5 10 RLUP** *has 5 in the new x-register*).

## RTN — or: Return Return_From_Subroutine

Returns to statement after the calling **GSB** if in a subroutine else returns to SIMION from called user program segment. Note: Use **EXIT** to force return to SIMION from program segment.

**SEED** or: **Random_Seed**

Uses the current contents of the x-register as a new seed to re-randomize the random number generator (*slow*). The x-register is unchanged. A value of 0.0 resets generator to its value at program start (*fast*). New random number generator used with SIMION 7.0 (*see Appendix E-15*).

**SEG** or: **Begin_Segment**
**SEG Name**

Begins a new data definition or program segment. **Name** must be one of the following: ***Define_Data, Initialize, Init_P_Values, Fast_Adjust, Efield_Adjust, Mfield_Adjust, Accel_Adjust, Other_Actions,*** or ***Terminate***(*e.g.* ***SEG Efield_Adjust*** *starts the efield adjust program segment*). The SEG command automatically inserts a leading **EXIT** command to force exiting of any preceding program segment. See discussion of program segments below for more details.

**SIN** or: **Sine**

Converts the contents of the x-register (***in radians***) to its sine (*e.g.* **1.570796 SIN** *becomes 1.0*).

**SQRT** or: **Square_Root**

Converts the contents of the x-register to its square root (*e.g.* **81 SQRT** *becomes 9.0*).

**STO** or: **Store**
**STO Name**

Stores the current contents of the x-register into the variable named **Name**. **Name** must be a currently active variable name that the user program segment is allowed to reference. **<u>If Name does not exist and it is not illegal, the compiler will create a temporary variable named Name</u>** (*e.g.* **STO TEMP1** *store x-register value to existing variable* **TEMP1** *or to a created temporary variable of that name*). See more information below concerning: Adjustable, static, reserved, and temporary variables.

**TAN** or: **Tangent**

Converts the contents of the x-register (***in radians***) to its tangent (*e.g.* **1.0 TAN** *becomes 1.557408*).

**X><Y** or: **X<>Y XY_Swap Swap_XY**

Exchanges the values in the current x and y registers (*e.g.* **1 2 X><Y** *puts 2 in y-register and 1 in x-register*).

| | | | |
|---|---|---|---|
| X=0 | or: | If_X_EQ_0 | If_X_Equals_0 |
| X!=0 | or: | If_X_NE_0 | If_X_Not_Equal_0 |
| X<0 | or: | If_X_LT_0 | If_X_Less_Than_0 |
| X<=0 | or: | If_X_LE_0 | If_X_Less_Than_Or_Equal_0 |
| X>0 | or: | If_X_GT_0 | If_X_Greater_Than_0 |
| X>=0 | or: | If_X_GE_0 | If_X_Greater_Than_Or_Equal_0 |

The six test commands above compare the x-register value to zero. If the selected test is true the next instruction following the test will be executed (**do if true**). Else the next instruction will be skipped (*e.g.* **5 X>=0 GSB MORE RTN** *results in calling subroutine* **MORE** *because 5 is greater or equal to 0*).

| | | | |
|---|---|---|---|
| X=Y | or: | If_X_EQ_Y | If_X_Equals_Y |
| X!=Y | or: | If_X_NE_Y | If_X_Not_Equal_Y |
| X<Y | or: | If_X_LT_Y | If_X_Less_Than_Y |
| X<=Y | or: | If_X_LE_Y | If_X_Less_Than_Or_Equal_Y |
| X>Y | or: | If_X_GT_Y | If_X_Greater_Than_Y |
| X>=Y | or: | If_X_GE_Y | If_X_Greater_Than_Or_Equal_Y |

The six test commands above compare the x-register value to the y-register. If the selected test is true the next instruction following the test will be executed (**do if true**). Else the next instruction will be skipped (*e.g.* **3 5 X<Y GSB MORE RTN** *results in executing RTN because 5 is not less than 3*).

## User Adjustable Variables

These are variables defined by three word **DEFA NAME VALUE** statements placed in the **Define_Data** segment (*at the top of a user program file*). Adjustable variables are read/write global variables that SIMION allows you to assign new values to before (***and during***) each **Fly'm**.

Adjustable variables are **globally visible**. **Thus ALL user program segments in ALL user program files that define an adjustable variable of the same name will actually reference the same adjustable variable.** The initial value of the adjustable variable will be that defined in the ***first*** user program file ***compiled*** that defined the adjustable variable (*always compiled in instance order*).

### Setting Adjustable Variables Before a Fly'm

When a Fly'm is initiated, user programs are automatically compiled and a list of adjustable variables are displayed to allow the user to change their values. You can change an adjustable variable's value before flying the ions. **Any changes at this point will be retained for the next Fly'm**. *Note:* Changes to adjustable variables made **while** ions are flying *(by you or user programs)* will be forgotten when the **Fly'm** terminates *(this allows program segments to communicate across rerun flights without permanently changing adjustable variables)*. Adjustable variables are reset to program defined values when a new **.IOB** file is loaded or the user programs are otherwise reset (*e.g. leaving and re-entering* **View**, *using the debugger, and etc.*).

### Selective Display of Adjustable Variables

Normally it would be desirable to display an abbreviated list of only those adjustable variables that the user really would want to change. If any adjustable variable name starts with a leading underscore (e.g. *_Acceleration_***voltage**) then only it and other adjustable variables with names ***beginning*** with an ***underscore*** character will (*normally*) be displayed.

However, it is recognized that there will be times when access to ***all*** adjustable variables may be desired. A button has been provided to shift between views of selected (*_OK*) and all adjustable variables (*ALL*). This button ***automatically*** appears at the top of the Adjustable Variables List Window when leading underscore names have been defined for adjustable variables. The default is **_OK** for displaying leading underscore variables only. However, if the button is depressed, the word **ALL** appears and all the adjustable variables will be immediately shown. ***Any changes to the current state of this option is conserved during the remainder of the SIMION session.***

> Note: The Adjustable Variables List Window has an *ON/OFF* button for user programs. The *ON/OFF* button allows you to turn user programs off during the current (*and only the current*) **Fly'm.** This is handy for applications when you may not want to use user programs in certain runs (*e.g. tuning*). **Hint: You can always define a dummy adjustable variable to gain access to the ON/OFF button.**

### Changing Adjustable Variables While Ions are Flying

SIMION also has an **AdjV** tab on the ***top of the View Screen*** that will be ***unblocked whenever adjustable variables are active*** during a **Fly'm**. Clicking this tab will give you access to adjustable variables (*via a panel screen*) while ions are flying (*selection slider provided if more than three variables to display*). By default, SIMION displays the adjustable variables it encountered when compiling your user programs. ***However***, if ***any*** adjustable variable name ***begins with an underscore*** (*e.g.* **_Damping**) ***only*** the ***adjustable variables compiled with leading underscores will be displayed*** (*allowing you to select your control variables*). ***Avoid*** displaying ***adjustable variables that the user programs write to*** (*change*), because ***SIMION will not display these changes***.

## *Static Variables*

These are variables defined by the three word **DEFS NAME VALUE** statements placed in the Define_Data segment (*at the top of a user program file*). Static variables are read/write global variables. **Static variables are very useful for record keeping within a user program segment and for communications between two or more active user program segments.**

Static variables are **globally visible**. **Thus ALL user program segments in ALL user program files that define a static variable of the same name will actually reference the same static variable.** The initial value of the static variable will be that defined in the ***first*** user program file ***compiled*** that defined the static variable (*compiled in instance order*).

SIMION 7.0 (*unlike 6.0*) prevents access to ***static*** variables and ***static arrays*** (*reading or writing*) from within the **Initialize**, **Init_P_Values**, and **Terminate** program segments. This is because the Initialize program segments are called for all ions before the first ion is flown. The Init_P_Values segments are also called before the first ion is flown. Moreover, the Terminate program segments are called only after ***all*** ions have flown. Since all static variables and arrays are ***always*** reset ***just before*** each ion (*or group*) is actually flown, static variables and static arrays cannot be passed information from Initialize, to Init_P_Values, or pass information to Terminate. The blocking of static variable and static array access in these three program segments was implemented to protect the user from coding errors (*your programs may have them*).

Unlike temporary variables the values of the static variables ***are not forgotten*** between calls to the user program segment. **Moreover, SIMION resets each static variable to its specified initial value at the beginning of <u>each individual or grouped</u> ion trajectory calculation (*flight or rerun*).**

## Array Variables

Both adjustable ***and*** static array variables are supported. The array feature is fully integrated into user programs including full program debugging and runtime error support.

### Array Elements and Addressing

Each array element is a double precision floating point number (8 bytes). Only single dimension arrays are supported (*as opposed to 2D or higher dimension arrays*). However, you can do your own index mapping computations if higher dimensions are wanted (*illustrated with **ASTO** and **ARCL** commands above*). Array elements begin with the index of ***<u>one</u>*** (*not zero as in C*). Thus the first element of a 100 element array has an index of 1 and the last element has the index of 100.

### Array Limits

Array sizes must be one or larger (*arrays are heap allocated so large sizes are legal - be careful*). The maximum number of unique Adjustable & Static arrays for ***all*** user program files is 200. The maximum number of Array Save and Load Commands for ***all*** user program files is 200.

### Lifetime of the Array Variables

***Adjustable*** array variables are initialized before ***<u>any</u>*** Initialize program segments are called. Their array values are retained ***<u>throughout</u>*** the period of ion flying (*while the **Fly'm** button is depressed*).

***Static*** array variables are initialized ***<u>immediately before</u>*** each ion (*or group*) begins to fly. Their array values are retained ***<u>only</u>*** until the end of the ion's (*or group's*) flight (*splat*). They have the same program segment access limits as described for normal static variables above.

### Array Initialization Options

All array elements of each defined adjustable and static array are ***<u>always</u>*** initialized to zero at the times described above. However, the user can also specify the name of an ASCII file containing array initialization data to be automatically loaded ***<u>after</u>*** the array has been pre-zeroed. If there are more values in the file than the defined array size, SIMION will only read the number of values required to fill the array. If there are fewer values in the file than the defined array size, SIMION will load the values provided starting with the first array element, and the remaining array elements will remain zeroed.

The array initialization files are free format ASCII. Numbers are separated by any number of spaces and/or commas (*commas are viewed as spaces*). Blank lines are allowed, and the ';' semicolon is recognized as the start of an inline comment (*as in user programming*). ***<u>Individual lines of initialization data must be less than 200 characters long</u>***.

The following is a legal though undesirable initialization file:

```
9 8 7 6 5 4 ;this is the first line
                        155.2

3.1415  2.0e-5
```

If there is illegal data in an array initialization file, SIMION will generate a runtime error help screen giving the file's name, the type of error, and its line number in the file.

### Array Commands

The following is a summary list of the available array commands. The commands are described in detail in the command summary above.

| | |
|---|---|
| **ADEFA** | Define Adjustable Array |
| **ADEFS** | Define Static Array |
| **ALOAD** | Load Array with data from an ASCII file |
| **ARCL** | Recall an Array element into a stack register |
| **ASAVE** | Save Array contents to an ASCII file |
| **ASTO** | Store a number on the stack into an Array element |

## *Temporary Variables*

Whenever the compiler encounters a **STO** statement with a variable name that ***doesn't*** match any currently defined variable (**of any type**) it automatically creates a temporary variable for the value. Temporary variables only retain their values during the execution of the program segment. **When the user program segment returns to SIMION all is forgotten.**

Temporary variables **must be defined** via a **STO** statement ***before*** they can be accessed. **Thus a RCL from an undefined variable (*assumed temporary*) will result in a fatal compiler error.**

Further, creating a temporary variable via a **STO** statement that is ***not*** referenced later via a **RCL** statement will ***also*** result in fatal compiler error. This prevents a miss-spelled reserved variable **STO** being made a temporary variable (*a mistake*) and thus introducing a hard-to-find program bug.

## *RESERVED VARIABLES AND THEIR FUNCTIONS*

User programs communicate to SIMION through special reserved variables. These reserved variables can be read (*user programs can input them with a* **RCL** *statement*) or written (*user programs can output to them with a* **STO** *statement*). SIMION limits the read/write access to reserved variables by program segment. This keeps you from exerting control at a bad time (***however***, *you are free to control things badly*). A table of reserved variables appears below:

There are three unit systems used with reserved variables. The first is the currently aligned workbench coordinates/orientation(*Variables using these coordinates/orientations have names ending with* ***_mm***). Variables using this unit system share the locations and orientations of the ***currently aligned workbench coordinates*** (*including* ***Align*** *button status*).

The second unit system is the ion's current instance's PA volume coordinates/orientation (*Variables using these coordinates/orientations have names ending with* ***_gu***). This is a reversible 3D transformation from the current workbench unit system into the 3D grid unit system and orientation of the ion's current instance.

The third unit system is the ion's current instance's PA Array coordinates ( *Variables using these coordinates have names ending with _Abs_gu*). This is a ***non-reversible transformation*** from 3D PA volume coordinates to the 2D or 3D coordinates of the actual potential array. For 3D arrays x, y, and z are converted into their absolute values. For 2D planar x and y are converted to their absolute values and z is set to zero. For 2D cylindrical x is converted to its absolute value, y and z are converted into r, which is stored in y, and z is set to zero.

| Variable Name | Use | Units | Read Access | Write Access |
|---|---|---|---|---|
| Adj_Elect00 to<br>Adj_Elect30 | Fast Adj Electrode[1]<br>Voltages | Volts | Fast_Adjust<br>Init_P_Values | Fast_Adjust<br>Init_P_Values |
| Adj_Pole00 to<br>Adj_Pole30 | Fast Adj Pole[1]<br>Mag Potentials | Mags | Fast_Adjust<br>Init_P_Values | Fast_Adjust<br>Init_P_Values |
| Ion_Ax_mm<br>Ion_Ay_mm<br>Ion_Az_mm | Ion's current<br>Acceleration<br>(WB Coordinates) | mm/micro $sec^2$<br>(WB Orientation) | Accel_Adjust<br>Other_Actions<br>Terminate | Accel_Adjust |
| Ion_BfieldX_gu<br>Ion_BfieldY_gu<br>Ion_BfieldZ_gu | Magnetic Field<br>at Ion's Location<br>(PA's Orientation) | Gauss<br>(PA's Orientation) | Mfield_Adjust | Mfield_Adjust |
| Ion_BfieldX_mm<br>Ion_BfieldY_mm<br>Ion_BfieldZ_mm | Magnetic Field<br>at Ion's Location<br>(WB Orientation) | Gauss<br>(WB Orientation) | Accel_Adjust<br>Other_Actions<br>Terminate | None |
| Ion_Charge | Ion's current charge | in units of<br>elementary charge | Any Prog Seg<br>except<br>Init_P_Values | Initialize<br>Other_Actions |
| Ion_Color | Color of Ion | 0-15 | Initialize<br>Other_Actions | Initialize<br>Other_Actions |
| Ion_DvoltsX_gu<br>Ion_DvoltsY_gu<br>Ion_DvoltsZ_gu | Voltage Gradient<br>at Ion's Location<br>(PA's Orientation) | Volts/grid unit<br>(PA's Orientation) | Efield_Adjust | Efield_Adjust |
| Ion_DvoltsX_mm<br>Ion_DvoltsY_mm<br>Ion_DvoltsZ_mm | Voltage Gradient<br>at Ion's Location<br>(WB Orientation) | Volts/mm<br>(WB Orientation) | Mfield_Adjust<br>Accel_Adjust<br>Other_Actions<br>Terminate | None |
| Ion_Instance | Current Instance | 1- max instance | Any Prog Seg<br>except<br>Init_P_Values | None |
| Ion_Mass | Ion's current mass | amu | Any Prog Seg<br>except<br>Init_P_Values | Initialize<br>Other_Actions |
| Ion_mm_Per_Grid_Unit | Min Current Scaling[2] | mm/grid unit | Any Prog Seg<br>except<br>Init_P_Values | None |
| Ion_Number | Ion's Number | 1- max ion | Any Prog Seg<br>except<br>Init_P_Values | None |

Table continued on next page:

| Variable Name | Use | Units | Read Access | Write Access |
|---|---|---|---|---|
| Ion_Px_Abs_gu<br>Ion_Py_Abs_gu<br>Ion_Pz_Abs_gu | Ion's current PA Array Coordinates | grid units (PA's Abs Coordinates) | Any Prog Seg except Init_P_Values | None |
| Ion_Px_gu<br>Ion_Py_gu<br>Ion_Pz_gu | Ion's current (PA's Coordinates) | grid units (PA's Coordinates) | Any Prog Seg except Init_P_Values | None |
| Ion_Px_mm<br>Ion_Py_mm<br>Ion_Pz_mm | Ion's current Workbench Coordinates | mm (WB Coordinates) | Any Prog Seg except Init_P_Values | Initialize Other_Actions |
| Ion_Splat | Ion Status Flag[3] | Flying = 0<br>Not Flying != 0 | Initialize Other_Actions | Initialize Other_Actions |
| Ion_Time_of_Birth | Ion's Birth Time | micro seconds | Any Prog Seg except Init_P_Values | Initialize Other_Actions |
| Ion_Time_of_Flight | Ion's current TOF[4] | micro seconds | Any Prog Seg except Init_P_Values | Other_Actions |
| Ion_Time_Step | Ion's Time Step[5] | micro seconds | All but Initialize and Init_P_Values | Tstep_Adjust |
| Ion_Volts | Electrostatic Pot at Ion's Location | Volts | Efield_Adjust<br>Mfield_Adjust<br>Accel_Adjust<br>Other_Actions<br>Terminate | Efield_Adjust |
| Ion_Vx_mm<br>Ion_Vy_mm<br>Ion_Vz_mm | Ion's current Velocity (WB Coordinates) | mm/micro sec (WB Orientation) | Any Prog Seg except Init_P_Values | Initialize Other_Actions |
| Rerun_Flym | Rerun Flym Flag[6] | NO = 0<br>YES = 1 | Initialize<br>Other_Actions<br>Terminate | Initialize<br>Other_Actions<br>Terminate |
| Trajectory_Image_Control | Controls Trajectory Image Viewing and Recording[7] | Value View Retain<br>0 YES YES<br>1 YES NO<br>2 NO YES<br>3 NO NO | Initialize<br>Other_Actions<br>Terminate | Initialize<br>Other_Actions<br>Terminate |
| Retain_Changed_Potentials | Controls restoring of changed potentials at end of Fly'm[8] | NO = 0 (default)<br>YES = 1 | None | Terminate |
| Update_PE_Surface | New PE Surface[9] | YES != 0 | None | Other_Actions |

## Notes on Reserved Variables

1 **Adj_Elect00** and **Adj_pole00** reserved variables support fast scaling capability of **.PA0** fast adjust files.

2 The value in the **Ion_mm_Per_Grid_Unit** reserved variable is ***normally*** the scaling of the ion's current instance. However, if the ion is currently in ***both*** electrostatic and magnetic potential array instances, the value stored is the smaller mm_per_grid_unit scaling (*e.g. for .25 and .30: .25 would be stored in variable*).

3 The **Ion_Splat** reserved variable has several possible SIMION generated values:

| | |
|---|---|
| 0 | keep-on-trucking |
| -1 | Hit electrode |
| -2 | Dead-in-water (*ion not moving and no forces on it*) |
| -3 | Outside workbench |
| -4 | Ion killed (*used in beam repulsion*) |

The **Other_Actions** segment can be used to monitor and change an ion's fate.

4 The value stored in **Ion_Time_of_Flight** depends on the program segment called:

| | |
|---|---|
| **Initialize** | Start time of next step |
| **Tstep_Adjust** | Start time of next step |
| **Fast_Adjust** | Start time of current test time step (*varies - Runge-Kutta*) |
| **Efield_Adjust** | Start time of current test time step (*varies - Runge-Kutta*) |
| **Mfield_Adjust** | Start time of current test time step (*varies - Runge-Kutta*) |
| **Accel_Adjust** | Start time of current test time step (*varies - Runge-Kutta*) |
| **Other_Actions** | Stop time of current step (*start time of following step*) |
| **Terminate** | Stop time of last step |

5 The value stored in **Ion_Time_Step** depends on the program segment called:

| | |
|---|---|
| **Tstep_Adjust** | Full Time step for next step |
| **Fast_Adjust** | Test time step (*varies - Runge-Kutta*) |
| **Efield_Adjust** | Test time step (*varies - Runge-Kutta*) |
| **Mfield_Adjust** | Test time step (*varies - Runge-Kutta*) |
| **Accel_Adjust** | Test time step (*varies - Runge-Kutta*) |
| **Other_Actions** | Time step of current step |
| **Terminate** | Last time step tried |

6 The **Other_Actions** segment can now read and write this variable. Setting this variable to 1 stops ion trajectory recording ***and*** erases any currently saved trajectories. Setting the variable to one (*1*) in a call to **Other_Actions** and then resetting it to Zero (*0*) on the next call to **Other_Actions** erases any currently saved trajectories. Toggling (*ON then Off*) this varialble in successive **Other_Actions** segments will not change data recording or the re-flying of ions (*use a Terminate segment to control re-flying*).

7 The **Trajectory_Image_Control** variable can be used to control viewing and retaining of ion trajectory images. It serves the same purpose as the **V** and **R** buttons (*new 7.0 features*) on the Normal Options screen. The least significant two bits are used to control these two functions. The truth table above shows the legal values of the variable and the results. SIMION always sets this variable according to the current **V** and **R** buttons' status. Clicking either the **V** or **R** button during a Fly'm will immediately change this variable back to the current values for **BOTH** the **V** and **R** buttons. ***Note: Changing the variable's values does change the V and/or R button's status.***

If Rerun is not active (*e.g. Rerun button not depressed*), the **Trajectory_Image_Control** variable can be used to selectively retain portions of ion trajectories (*in the trajectory image file*). Set bit 0 (*Retaining bit*) of this variable in an **Other_Actions** program segment to suppress trajectory saving. This assumes the **R** button is depressed (*retaining by default*). If the **R** button is not depressed (*not retaining by default*), you would clear bit 0 of this variable to start trajectory image retention, and then set it to suppress as desired.

Bit 1 (*Viewing bit*) controls the viewing of trajectories. This bit can be set and cleared under program control to select what portions of the trajectories are actually displayed during the Fly'm itself. Its function is completely independent of either bit 0 (*Retain bit*) or the status of the **Rerun** button.

Note: SIMION draws the current time step trajectory vector (*2 point*) immediately ***after*** the **Other_Actions** program segment executes. Marks are actually drawn on the following time step so you will have to keep saving trajectories one extra time step to get the mark.

8 The **Retain_Changed_Potentials** variable is used to retain potential array changes caused by either the **Init_P_Values** program segment or the actions of the **Update_PE_Surface** variable beyond the end of the current Fly'm. Normally SIMION restores any changed potentials to their original values at the end of each Fly'm. Storing a non-zero value to this variable in any **Terminate** segment will retain **all** potential changes to **all** active arrays.

9 Storing a ***non-zero value*** in **Update_PE_Surface** (*e.g. 1*) requests a PE surface update (*can only be called from* **Other_Actions** *segment*). If there is a **Fast_Adjust** ***program segment active*** and ***PE view mode is currently active***, the **Fast_Adjust** program segment will be called after **Other_Action** exits. The fast adjust potentials it returns will be used to fast adjust the entire potential array and its PE surface will be redrawn. **<u>This is an excellent way to make the PE surfaces undulate</u>**. See user program demos for examples of how to use it effectively: **_BUNCHER**, **_RFDEMO**, **_TRAP**, and **_TUNE**.

## *Numerical Constants*

Whenever the compiler encounters a number when it is looking for a command it will assume it is a constant to be loaded into the x-register (*via* **RCL** *constant*). In order to save space, it keeps an array of unique numerical constants. Multiple references to the same number will automatically be translated into multiple references to the same numerical constant.

## *Compiler Limits*

The debugging compiler outputs a compilation summary of the resources used. The following memory limits apply to user programs:

1. Program memory is limited to 5,000 commands ***per user program file*** (*e.g.* **.PRG** *file*).
2. The maximum number of **<u>unique</u>** adjustable variables for ***<u>all</u> active user program files*** is 200.
3. The maximum number of **<u>unique</u>** static variables for ***<u>all</u> active user program files*** is 200.
4. The maximum number of **<u>unique</u>** Adjustable + Static arrays for ***<u>all</u> active user program files*** is 200.
5. The maximum number of **<u>unique</u>** Array Save + Load Cmds. for ***<u>all</u> active user program files*** is 200.
6. The maximum number of **<u>unique</u>** numerical constants ***per user program file*** is 200.
7. The maximum number of **<u>unique</u>** messages for ***per user program file*** is 100 (*50 character*).
8. The maximum number of **<u>unique</u>** temporary variables in a ***single user program segment*** is 200.
9. The maximum number of **<u>unique</u>** entry point labels in a ***single user program segment*** is 200.

## *Details of User Program Segments*

User program segments (*if defined*) are called like subroutines from within SIMION. Figure I-1 below contains a simple flow diagram of ion trajectory calculations. The diagram shows where user programs are compiled, adjustable variables changed, static variables reset, and the various user program segments called. Take the time now to carefully study the general flow of events. It is important that you clearly understand where and when each user program segment is called if you expect to make creative (*or effective*) use of user programming within SIMION.

***Each user program segment is implemented as a monitor/modifier to the normal course of events.*** That is, SIMION calculates the next time step to use for an ion or group of ions ***<u>and</u> then*** calls the **Tstep_Adjust** program segment (*if it exists*). The **Tstep_Adjust** program segment can then monitor and perhaps change the proposed time step. This gives you complete freedom to monitor, modify, or substitute your own values for important items like: Time step; electrostatic and magnetic fields; accelerations; and even ion mass, charge, and etc.

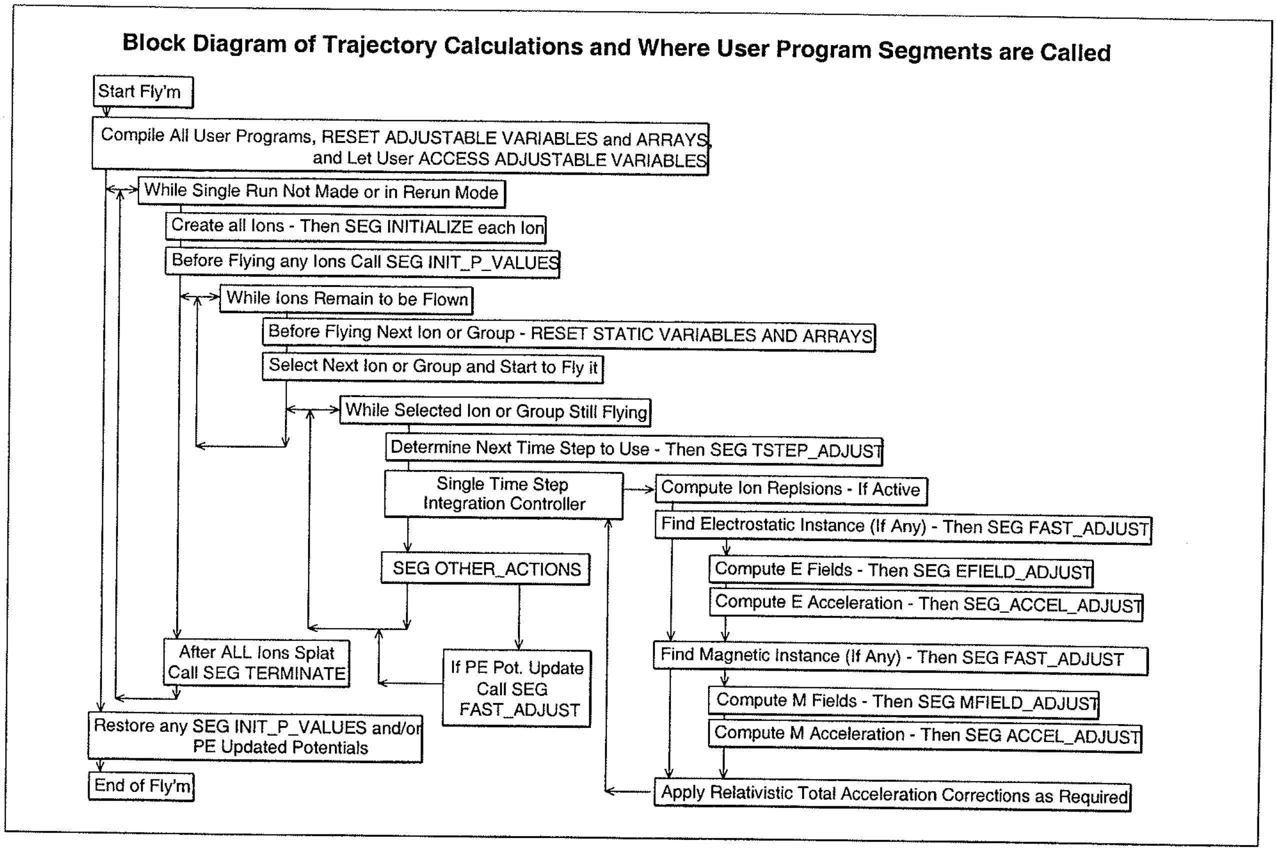



**Figure I-1 Ion trajectory calculation block diagram showing when variables and arrays are reset and where the various user program segments are called from.**

The reserved variables **Ion_Time_of_Flight** and **Ion_Time_Step** take on different values depending on what program segment they are referenced from. The **Fast_Adjust**, **Efield_Adjust**, **Mfield_Adjust**, and **Accel_Adjust** program segments are called multiple times during a Runge-Kutta integration step. The value of **Ion_Time_of_Flight** is the TOF in microseconds at the ***start*** of the specific Runge-Kutta term's step. The **Ion_Time_Step** is the specific Runge-Kutta term's time step. You can use these values to get the desired TOF for your uses. ***For example, if you need the middle TOF of the term's time step load the Ion_Time_of_Flight and add half the Ion_Time_Step to it.*** See the notes on reserved variables above for more information.

## The Initialize Program Segment

The **Initialize** program segment (*if active*) is called after each ion has been initialized for the next **Fly'm** (*or rerun*). At this point you have the option of changing the following parameters: **Ion_Charge**, **Ion_Color**, **Ion_Mass**, Ion's WB position, **Ion_Splat**, **Ion_Time_of_Birth**, Ion's WB velocity, and **Rerun_Flym**. **Initialize** segment can output **Message** and **R/S** commands. It is useful for supporting the rerunning of trajectories under program control (*looping back from the* **Terminate** *segment via the* **Rerun_Flym** *reserved variable*). ***Use adjustable variables of adjustable arrays to communicate between reruns. Access to static arrays and static variables is not allowed as they have undefined values when the program segment is called.***

```
Defa Mass 100                              ;Mass to use set by user and Terminate

Seg Initialize                             ; beginning of segment
        RCL Ion_Vz_mm                      ; recall starting velocity
        RCL Ion_Vy_mm
        RCL Ion_Vx_mm
        >P3D                               ; convert to polar 3d
        RCL Ion_Mass                       ; recall initial mass
        x><y >KE                           ; get ion's ke
        x><y RLUP RCL Mass STO Ion_Mass    ; substitute current mass
        x><y >SPD x><y RLUP                ; convert back to speed
        >R3D                               ; get new velocity with same ke & new mass
        STO Ion_Vx_mm RLUP                 ; save new starting velocity
        STO Ion_Vy_mm RLUP
        STO Ion_Vz_mm
```

## The Init_P_Values Program Segment

A program segment called **Init_P_Values** can initialize entire fast adjust potential arrays (*e.g. .PA0*) **before** flying **any** ions. SIMION can also fast adjust a ***copy*** of the array points that neighbor the ion's current location (*the exception is **Update_PE_Surface***) as the ion(s) fly via the **Fast_Adjust** program segment (*discussed below*). However, if potentials won't be changed except between successive fly'ms in a series (*e.g. auto-tuning*), it is more efficient to change the entire **.PA0** array's potentials once ***before*** the ions are flying to avoid the overhead caused by successive calls to seg **Fast_Adjust**. The actual time savings will be a tradeoff between array size and the number of ions flown. Small arrays with large numbers of ions will probably show the biggest improvement.

The **Init_P_Values** program segment has been provided to do this. When SIMION starts a fly'm or a rerun it first initializes **ALL** ions (*calling **Initialize** segments as appropriate*). It **next** calls **all** the **Init_P_Values** segments defined in the all the active user program files (*.**PRGs** files*). **Then** SIMION initializes static arrays and variables just before flying each ion or group.

Note: Unlike *any* other program segment, ions do ***not*** have to be in the instance to have the **Init_P_Values** program segment called. ***This also means that ion and instance context have no meaning within this program segment***. SIMION will flag a compiler error if you try to access **any** ion or instance related reserved variable. The ***only*** reserved variables that can be accessed are **Adj_Elect00 – Adj_Elect30** and **Adj_Pole00 – Adj_Pole30**. Moreover, use of an instance related coordinate transformation (*e.g. >ARR, >PAC, >PAO, >WBC, and >WBO*) will flag a compiler error. Access to static arrays and static variables is **not** allowed as they have undefined values when the program segment is called (*as with **Initialize** and **Terminate***).

SIMION normally restores any changed potentials to their pre Fly'm values at the end of the Fly'm. However, you have the option of retaining these changed potentials with a reserved variable called **Retain_Changed_Potentials**. If you ***save*** a non-zero value to this variable from within any **Terminate** program segment SIMION will not restore the changed potentials. This works for both the **Init_P_Values** program segment changes and for **Update_PE_Surface** induced potential changes.

```
Defa Tune_Voltage 100                      ;tuning voltage to use

Seg Init_P_Values                          ; beginning of segment
        RCL Tune_Voltage
        STO Adj_Elect01                    ; store tune_voltage in fast adjust electrode one
        Exit
```

## The Tstep_Adjust Program Segment

The **Tstep_Adjust** program segment is used to monitor/adjust the **Ion_Time_Step** (*in micro seconds*) to use with the selected ion. ***Note: The time step is really a requested time step***. Other circumstances like other ions (*grouped ion flying*), delayed time-of-birth, binary boundary approaches of the ion(s) can result in a ***shorter*** time step being used. However, if you write your program segment to be persistent you can hit exact time-of-flight points. The example below serves as an illustration:

```
Defa  Start_Time  100                         ;starting time to use

Seg Tstep_Adjust                              ; beginning of segment
        RCL Start_Time
        RCL Ion_time_of_Flight
        X>=Y  EXIT                            ; exit if ion will be at or beyond start time
        RCL Ion_Time_Step +
        X<=Y EXIT                             ; exit if time step less or equal to that needed
        RCL Start_Time
        RCL Ion_time_of_Flight -
        STO Ion_Time_Step                     ; else shorten time step to what is just right
```

## The Fast_Adjust Program Segment

The **Fast_Adjust** program segment is only legal with **.PAO** fast adjust potential arrays. This program segment allows you to fast adjust ***and/or*** fast scale (*via* ***Adj_Elect00***) the potentials of adjustable electrodes or poles as the ion flies. This is very useful for all sorts of simulations. **Note: Only change via a Fast_Adjust program segment those electrode potentials you need to change as the ions fly (*faster and saves RAM*).**

In order to execute as fast as possible, SIMION only loads a copy of each ***needed*** fast adjust (*or fast scale -* ***.PA_***) electrode solution PA file (*because the* **Fast_Adjust** *program segment changes their electrode's/pole's potential*) into memory (*e.g.* **.PA1**). If your computer doesn't have enough physical RAM (*RAM needed = 8 bytes * array size * electrodes stored*) virtual memory will be used (*as available*). ***If not enough memory is available (real or virtual) SIMION will abort the trajectory run (you may have to increase your virtual size)***. Once these files have been loaded SIMION will try to avoid reloading them (*potentially a slow process*).

The example shows a simple **Fast_Adjust** program segment (*Hint: Use* **Tstep_Adjust** *segment above to create a precise turn-on time*):

```
Defa  Start_Time  100                         ; starting time to use
Defa  AC_Voltage 500                          ; Voltage to use
Defa  Omega 25                                ; angular frequency

Seg Fast_Adjust                               ; beginning of segment
        0.0  STO voltage                      ; assume zero volts
        RCL Ion_time_of_Flight
        RCL Start_Time
        X>Y  goto done                        ; skip ac if start time > TOF
        - RCL Omega * SIN                     ; sin(omega * (TOF - Start_Time))
        RCL AC_Voltage *  STO voltage         ; ac voltage to use
LBL done
        RCL voltage
        STO  Adj_Elect01                      ;adjust electrode number one
```

## The Efield_Adjust Program Segment

The **Efield_Adjust** program segment can be used to monitor and set the **Ion_Volts**, **Ion_Dvoltsx_gu**, **Ion_DvoltsY_gu**, and **Ion_DvoltsZ_gu** electrostatic potential and fields. ***Note: This user program segment can only be used with an electrostatic potential array***. Even though your potential array may be 2D, the potentials and fields ***supplied/required*** are the full **3D** fields produced by your array as projected as a 3D object in the workbench. The field gradients are in volts per grid unit and are PA oriented (*to simplify your task*). SIMION scales and orients these fields into workbench coordinates after program segment exit.

```
Defa  Start_Time                          ; time to start field
Defa  AC_Voltage 500                      ; voltage to use
Defa  Omega 25                            ; angular frequency

Seg Efield_Adjust                         ; beginning of segment
        RCL Ion_Time_of_Flight            ; test to see that field is on
        RCL Start_Time -
        x>0 goto running                  ; jump to running if field is on
        0 STO Ion_Volts                   ; return zero field
        STO Ion_DvoltsX_gu
        STO Ion_DvoltsY_gu
        STO Ion_DvoltsZ_gu
        EXIT
lbl running
        RCL Omega * SIN                   ; sin(omega * (TOF - Start_Time))
        RCL AC_Voltage *                  ; ac voltage to use
        STO Ion_DvoltsZ_gu                ; set gradient in z
        RCL Ion_Pz_gu *                   ; compute voltage
        STO Ion_Volts                     ; save voltage
        0 STO Ion_DvoltsX_gu              ; set x and y field components to zero
        STO Ion_DvoltsY_gu
```

## The Mfield_Adjust Program Segment

The **Mfield_Adjust** program segment can be used to monitor and set the **Ion_BfieldX_gu**, **Ion_BfieldY_gu**, and **Ion_BfieldZ_gu** magnetic fields (*gauss*). ***Note: This user program segment can only be used with a magnetic potential array***. Even though your potential array may be 2D, the potentials and fields supplied/required are the full 3D fields produced by your array as projected as a 3D object in the workbench. The field gradients are in gauss and are PA oriented (*to simplify your task*). SIMION orients these magnetic fields into workbench orientation after program segment exit.

```
Defa  Start_Time                          ; time to start field
Defa  Gauss 5000                          ; magnetic field to use

Seg Mfield_Adjust                         ; beginning of segment
        0 STO Ion_BfieldX_gu              ; assume field is off
        STO Ion_BfieldY_gu
        STO Ion_BfieldZ_gu
        RCL Ion_Time_of_Flight            ; test to see that field is on
        RCL Start_Time
        x>y EXIT                          ; field is off
        RCL Gauss  STO Ion_BfieldZ_gu     ; create field in z direction
```

## The Accel_Adjust Program Segment

The **Accel_Adjust** program segment can be called by electrostatic ***and/or*** magnetic potential array instances. In either case, SIMION computes the electrostatic or magnetic accelerations (*as appropriate*) and ***adds these components to the total ion's acceleration*** and then calls the appropriate **Accel_Adjust** program segment. The program segment can then input and modify the ***ion's total acceleration calculated <u>at this point</u>*** as appropriate.

From the flow diagram (*above*) note the ***order of the ion's total acceleration calculation***:

1. Ion Repulsion Accelerations (*if any*)
2. Electrostatic Accelerations(*if any*) added (*E array's seg **Accel_Adjust** called*)
3. Magnetic Accelerations(*if any*) are added (*M array's seg **Accel_Adjust** called*)

***Note: Relativistic corrections are applied <u>after</u> the total non-relativistic acceleration has been computed. Thus the acceleration seen by Accel_Adjust program segments is <u>always</u> non-relativistic***. However, the ion's acceleration components as seen by the **Other_Actions** program segment contain the relativistic corrections at the ion's new location (*after the time step has been applied*). See Appendix E for information on how relativistic corrections are performed.

Trick 1: To isolate the electrostatic acceleration in electrostatic PA's **Accel_Adjust** segment ***when charge repulsion is active*** (*total acceleration contains both at that time*), create an **Efield_Adjust** segment in the **.PRG** file that saves the acceleration components to three static variables (*total acceleration contains only ion repulsion accelerations at that point*). Now in the **Accel_Adjust** segment recall the current total acceleration and subtract the ion repulsion acceleration from it (*stored in the static variables*). A similar trick could be used to isolate magnetic components fròm the total acceleration in a magnetic PA's user programs.

Trick 2: If you want to add viscosity effects using the integration stabilization ***below*** it should be applied to the true total acceleration. The proper location for the **Accel_Adjust** segment would be in the electrostatic PA's user program file if only electrostatic fields are active or in the magnetic PA's user program file if magnetic fields are active (*too or only*).

## An Improved Accel_Adjust Segment for Viscosity

The problem with viscosity is that it has an exponential transient decay of acceleration. When the damping term is very small (***large** time constant*) the Runge-Kutta integration works just fine. However, when the damping is high (***small** time constant*) the Runge-Kutta integration can overestimate the acceleration badly. SIMION's CV self protection (*when quality > 0*) detects this problem and chops the time step. While the computation is salvaged in this manner the time step can be so small that it may take 30 minutes or longer to fly a single ion.

The trick is to give the Runge-Kutta system what it really wants: ***An estimate of the <u>average</u> acceleration within the test time step***. It can be shown that the average acceleration can be computed by multiplying the initial total acceleration including viscosity (*at the beginning of the test time step*) by the following factor:

**factor = (1 - e^(-tstep * damping))/(tstep * damping)**

```
; This fixed Accel_Adjust program segment adds a Stokes' Law Viscosity Effect
; It is an example of the external problem fix (in the honored NASA Hubble tradition)
; It also demonstrates the real power of SIMION's user programming

Define_Adjustable Viscous_Damping 0          ; adjustable variable Viscous_Damping

Begin_Segment Accel_Adjust                   ; start of accel_adjust program segment
        Recall Ion_Time_Step x=0 Exit        ; exit if zero time step
        Recall Damping x=0 Exit              ; exit if zero damping
        * Store nt                           ; number of time constants in step
        chs e^x 1 x><y -                     ; (1 - e^(-nt))
        Recall nt / Store factor             ; divide by nt store as factor

        Recall Ion_Ax_mm                     ; recall current x acceleration (mm/usec^2)
        Recall Ion_Vx_mm                     ; recall current x velocity (mm/sec)
        Recall Viscous_Damping               ; recall the viscous damping term
        Multiply                             ; multiply times x velocity
        Subtract                             ; and subtract from x acceleration
        Recall factor *                      ; multiply by exponential averaging factor
        Store Ion_Ax_mm                      ; return adjusted value to SIMION

        Recall Ion_Ay_mm                     ; recall current y acceleration (mm/usec^2)
        Recall Ion_Vy_mm                     ; recall current y velocity (mm/sec)
        Recall Viscous_Damping               ; recall the viscous damping term
        Multiply                             ; multiply times y velocity
        Subtract                             ; and subtract from y acceleration
        Recall factor *                      ; multiply by exponential averaging factor
        Store Ion_Ay_mm                      ; return adjusted value to SIMION

        Recall Ion_Az_mm                     ; recall current z acceleration (mm/usec^2)
        Recall Ion_Vz_mm                     ; recall current z velocity (mm/sec)
        Recall Viscous_Damping               ; recall the viscous damping term
        Multiply                             ; multiply times z velocity
        Subtract                             ; and subtract from z acceleration
        Recall factor *                      ; multiply by exponential averaging factor
        Store Ion_Az_mm                      ; return adjusted value to SIMION
Exit                                         ; exit to SIMION (optional statement)
```

## The Other_Actions Program Segment

The **Other_Actions** program segment (*if active*) is called after each ion's time step. At this point you have the option of changing the following parameters: **Ion_Charge**, **Ion_Color**, **Ion_Mass**, Ion's WB position, **Ion_Splat**, **Ion_Time_of_Birth**, and Ion's WB velocity. This program is most useful for changing ion parameters (*e.g. mass or charge*) during its flight.

The **Other_Actions** segment can also read and write the **Rerun_Flym** reserved variable. Setting this variable to 1 stops ion trajectory recording ***and*** erases any currently saved trajectories. Setting the variable to one (*1*) in a call to Other_Actions and then resetting it to Zero (*0*) on the next call to Other_Actions erases any currently saved trajectories. Toggling (*ON then Off*) this variable in Other_Actions segment will not change data recording or the re-flying of ions.

Note: You can also perform binary boundary approaches (*of all types*) with **Other_Actions** and **Tstep_Adjust** segments working together. Each time the **Other_Actions** is called it could look for a boundary. If the boundary was crossed (*e.g. velocity*) it would restore ion's parameters at the end of last step (*stored in static variables by* **Initialize** *and* **Other_Actions** *just before they exit*) and flag a halving of the time step to the **Tstep_Adjust** segment. ***Of course there would have to be some minimum time step looping exit limit or the program would lock up***.

```
; For use with Tstep_Adjust segment above to neutralize ions at Start_Time
Defa Start_Time 100                          ; time to make change

Seg Other_Actions                            ; beginning of segment
        RCL Start_Time                       ; time to make change
        RCL Ion_Time_of_Flight               ; ion's TOF
        x!= 0 EXIT                           ; exit if not at switch point
        RCL Ion_PZ_mm                        ; get ion's position
        RCL Ion_PY_mm
        RCL Ion_PX_mm
        RCL Ion_Number                       ; get ion's number
                                             ; output ion data at neutralization point
        MESS ; Ion # Neutralized at: x = #, y = #, z = #
        BEEP                                 ; sound a beep too
        1 STO Ion_Color                      ; switch ion's color to red
        0 STO Ion_Charge                     ; neutralize ion
        mark                                 ; mark event
```

## The Terminate Program Segment

The **Terminate** program segment (*if active*) is called ***after <u>all</u>*** ions have stopped flying in the current flying cycle. At this point you have the option of changing the **Rerun_Flym** reserved variable. Setting this variable to 1 depresses the **Rerun** button and the ions are re-flown. Setting **Rerun_Flym** to 0 turns off the **Rerun** button and the current **Fly'm** is terminated. Note: If the **Rerun_Flym** variable is not changed, rerunning retains its current status (***that of the*** **Rerun** ***button***). ***Note: Use Adjustable variables to communicate between reruns***. <u>*Access to static arrays and static variables is not allowed as they have undefined values when the program segment is called*</u>.

***<u>Important:</u>*** When the **Rerun** Button is depressed (*by you or* **Rerun_Flym**) ion trajectories are not saved (*remembered for redrawing*). ***Trick***: To save the trajectories in the last run clear the **Rerun_Flym** variable with the Initialize segment ***at the start of the last run***. ***Also***: ***Data recording to a <u>file</u>*** is blocked ***<u>if and only if</u>*** the **Rerun** button is depressed ***<u>before</u>*** the **Fly'm** button is clicked.

```
; For use with Initialize segment above as a looping demo
Defa Mass 100                                ; Mass to use set by user and Terminate
Defa Del_Mass 1                              ; delta mass between runs
Defa N_Runs 10                               ; number of runs to make

Seg Terminate                                ; beginning of segment
        RCL Ion_Number
        1 X!=Y EXIT                          ; exit if not ion number one
        1 STO Rerun_Flym                     ; set rerun by default
        RCL Mass RCL Del_Mass + STO Mass     ; Mass += Del_Mass
        RCL N_Runs 1 - STO N_Runs            ; dec n_runs by one
        x>0 EXIT                             ; rerun if runs remain
        0 STO Rerun_Flym                     ; terminate Fly'm after this run
```

## User Program Demos

There are several user program demonstration sub directories. ***The files in these subdirectories were shipped compressed.*** To maximize compression of .PA files all non-electrode points were set to zero. ***Thus you must load, refine and save specific arrays before you can execute the demos successfully.*** The **README.DOC** files in each of these sub directories contain instructions on what to do. *Note: These demos merely demonstrate samples of simplified user programming. They are most likely not the most appropriate algorithms for the task they demonstrate. That task is YOUR responsibility*:

| | |
|---|---|
| **_BUNCHER:** | Ion bunching demo |
| **_DRAG:** | Simple Stoke's Law viscosity demo |
| **_ICRCELL:** | Full 3D ICR Cell with external ion injection modeled |
| **_QUAD:** | Quadrupole demo with 3D modeling of entrance and exit regions |
| **_RANDOM:** | Random ion generator demo |
| **_RFDEMO:** | Simple demos of various ways to generate RF effects |
| **_TRAP:** | Ion Trap Demo with damping, ion repulsion, and undulating PE surfaces |
| **_TUNE:** | Auto-focusing demo for a simple lens |

## Creating and Testing User Programs

User program files can be created with EDY (*or your favorite editor*) as a simple ASCII file. The actual creation of a user program file is normally done outside SIMION. However, you can click the GUI File Manager button on the Main Menu Screen and then click the **Edit** button in the file manager to gain direct access to EDY from within SIMION. ***Remember that all user program files must have the same name as their potential array and a .PRG extension. Moreover, all user program files must be in the same project directory as the potential arrays they support.***

### Running Another Editor From Within SIMION

The Debugger uses the EDY editor by default. If you prefer another editor, see Appendix H on the EDY editor to see how to use environmental variables to access other editors from within SIMION.

### Testing User Programs with SIMION's Debugger (Figure I-2)

Whenever SIMION loads a potential array it automatically compiles any associated user program file. To test or debug the user program segments associated with a particular potential array, click the potential array's button within the active PA window on the Main Menu Screen. If the selected potential array has a user program file the **Test & Debug** button will be unblocked. To test or debug these user program segments click the **Test & Debug** button.

Note: There is also a way to access the debugger from within the **View** function. Click the **PAs** tab, select the desired instance, and then click the **Dbug** button. ***Access to the* Dbug *button will be blocked if the instance's PA has no* .PRG *file or ions are currently flying.***

### Getting the Lay of the Land

The debugging compiler screen is composed of a collection of control objects above an activity display screen (*see Figure I-2 below*). There are four groups of objects: Key code support, compiler controls, file access, and debugging controls.

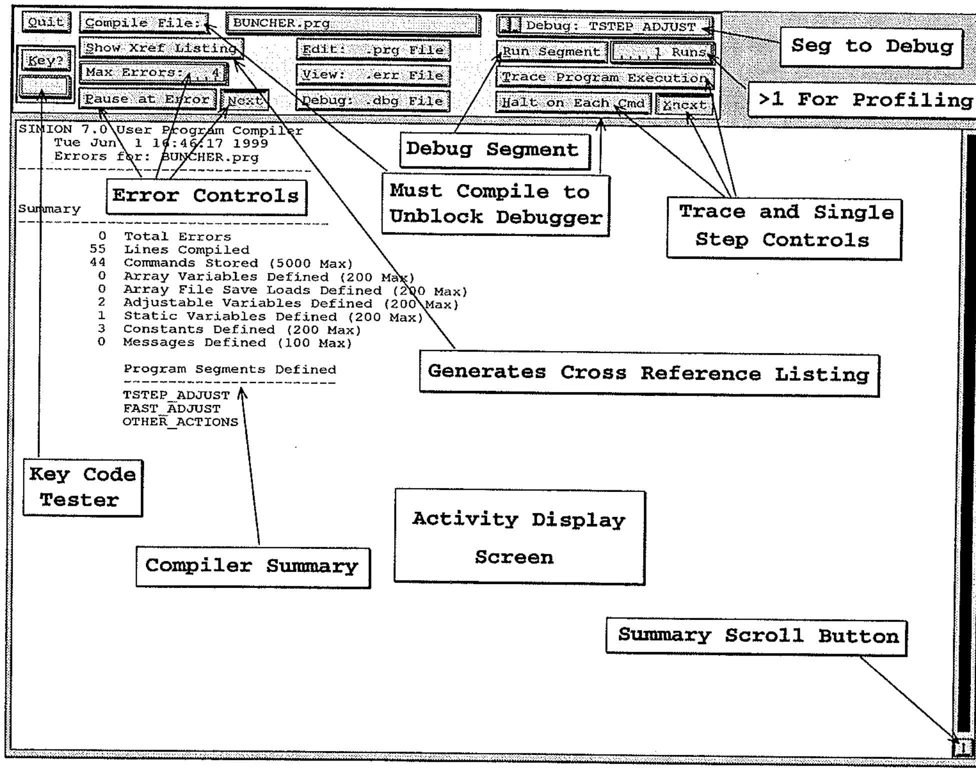



**Figure I-2 SIMION's User Program compile and debug facility**

## Key Code Support

Note: This feature is used to determine the key codes that **KEY?** and **R/S** commands put in the x-register. The **Key?** button and the display object below it provide key code support. To determine the key code of any keyboard key, click the **Key?** button and then press the desired key (*or key combination, e.g.* **<Ctrl A>**). The key code will appear in the display object.

## Compiler Controls

The next column of objects to the right contains the compiler controls. They are used to test compile the current **.PRG** file. The compiler outputs to a **.ERR** file. This file is kept ***either*** if there are errors ***or*** if the **Show Xref Listing** button is depressed.

The **Compile** button is used to start the test compiler. **Note: You must test compile successfully (*no errors*) <u>before</u> access to the debugger will be unblocked.**

The **Show Xref Listing** button is used to generate a cross reference listing in the **.ERR** file. This is useful to see how your user program was compiled. ***It also provides the code addresses so you can locate problems in your source when run-time errors are generated.***

The last three objects are used for compilation error control. The panel object is used to set the maximum number of errors allowed before test compilation aborts. Setting the value to one aborts compilation on the first error. You can then correct the source file (**Edit** *button*) and try again. The two remaining buttons are used to activate a pause at each error and the compile to the next error options.

### File Access

The the center column of objects contains three buttons that allow you to access specific files with EDY. The **Edit** button accesses the current **.PRG** file. When you return from editing the source file (**.PRG**) SIMION assumes that you probably changed something, and automatically blocks access to the debugger until you successfully test compile again.

The **View** button is used to edit the **.ERR** file. **This file is only kept (*stored*) if there is an error or the Show Xref Listing button was depressed during a test compile.**

The **Debug** button is used to edit the **.DBG** debugger output file. Each time the debugger runs it creates a **.DBG** file containing the debugger's output.

### Debugging Controls

The right column contains objects that are used for running the debugger. ***These controls are blocked until a successful test compilation has been made.*** The selector object (*on top*) allows you to select the program segment to debug.

The **Run Segment** button is used to run the debugger on the selected program segment. When the debugger executes it compiles the selected segment; allows you to set the values of all adjustable, static, and reserved variables ***used***; and then runs the program segment.

The type of run is controlled by the remaining four control objects. The **Runs** panel object accesses the run-time profiler ***if and only if*** more than one run is requested. This allows you to determine how fast your user program segment executes. Be sure to use a large enough number of runs to get a relatively accurate set of estimates. This feature is very useful if you need to optimize the performance of a user program segment.

The **Trace Program Execution** button puts the debugger in trace mode. Each command that executes produces a line of trace information containing the command, its address, function, and the contents of the lowest four registers (*x, y, z, and t*).

If the **Halt on Each Cmd** button is depressed the debugger will run in trace mode and halt after each command is executed. Click the **Xnext** button to execute the next command. These controls are useful for single step execution of a program segment.

## Runtime Errors

The user program run-time system is designed to catch most execution errors (*e.g. dividing by zero*). When a run-time error is detected (*whether in the debugger or when flying ions*) the run-time system will halt the execution, display the type of error including the offending command's address, and abort any further execution of user program segments (*e.g. abort the* **Fly'm**). ***Use a cross-reference listing to find the location of the error in your source.***

## Endless Loop Lockups

The user program run-time system is designed to allow you to exit an accidental endless loop lockup. If you are flying ions just hit the **Esc** key and the loop will be exited and the **Fly'm** aborted.

If you loop lock in the debugger you can also hit the **Esc** key. ***However, there is also a more useful approach.*** Click on the **Halt on Each Cmd** button (*YES, the mouse works in the debugger even in a locked loop*). The debugger will instantly switch to single step trace mode. Now click the **Xnext** button to step through the execution and see just where and under what conditions the program segment is loop locked. When you've found it, just hit the **Esc** key to stop the debugger.