Steve Clark, AIA

Contact Steve Clark at: clarks@mindspring.com
or 011-506-271-0241 (voice & fax)
or SJO 1219, PO Box 025216, Miami, FL. 33102-5216

Lesson 1
What is a Sigmac?

What is Sigmac?

Literally, it could mean **Sig**ma's **mac**ro or **SigmaC** language (C-like). It's a deeply guarded SDI secret. According to the grapevine, sigmacs are actually interpreted within ARRIS.

ARRIS CAD is divided into 3 levels of code:

1) 'graphics.exe' CAD Engine private
2) Library of $Utilities private
3) Applications public

The first 2 levels are written in C and the source code is private, while the last level (Applications) is written in Sigmac and is made public (most of it). The application level is a collection of (system) sigmacs, so look at them, study them, then you can fix them.

The Sigmac language consists of *syntax*, a *toolbox* and *logic*. *Syntax* is the grammar of the language, consisting of *symbols* and *keywords* that makeup executing statements. The *toolbox* is the set of application *sigmacs* and *functions*. Functions are powerful and complex, affectionately referred to as *dollar utilities* (they all begin with "$"). $utilities expect arguments in a strict order and may or may not return data. *Logic* is the *flow* (or control) in which statements are executed.

In the beginning, stay with chaining together sigmacs (or commands). Don't worry about $utilities now, we'll introduce a few as we go along.

The sigmac process consists of:

1) Sigmac Source File (ff) - where you write the code in readable ASCII text. The sigmac with the name "ustart" will execute automatically when ARRIS is initiated (as long as it is archived in "file.sm" or "user.sm"). The "ff" extension will tag this ARRIS file as a sigmac *source* file.

The *source* file is **very important**, without it - you will not be able to modify your sigmac in the future.

2) Compiler - where the source code is converted into machine code. The compiler will report syntax errors.

3) Machine Code File (gg) - what is produce after a successful compilation. The "gg" extension will tag this ARRIS file as a sigmac *machine code* file.

4) Archiver - archives (places) the gg file into a sigmac library (sm). Once the "gg" file is archive the harddisk copy can be deleted (only necessary in UNIX).

5) Sigmac Library (sm) - a collection of individual sigmacs. The *sigmac library* is what you load into ARRIS. There are 2 sigmac libraries that will load automatically when ARRIS is initiated, "file.sm" and "user.sm". The "sm" extension will tag this ARRIS file as a *sigmac library* file.

6) Debugging - a process in which you slowly pull your hair out looking for logic errors. Once you feel you have figured out what went wrong, repeat, rinse, repeat rinse, ...

You may use any text editor you are comfortable with to write Sigmacs. Sigma Design provides a beta version of ADE (ARRIS Developement Environment) in the Windows world. In UNIX, you're stuck with a superior product (IMHO) called the VI editor. In fact, I found a very good VI editor for the windows world, called "Lemmy" from www.softwareonline.org for about $40. It combines the best of the VI editor with the GUI cut&paste world.

Lesson 2
First Sigmac - "Hello World"

A sigmac consists of a series of *statements* and their *flow* of execution. There are many different types of statements to discuss, we'll start with the basics and go from there.

Write It

Open any text editor and type:

```
001     /* hello.ff     sc      03.15.2000      ARRIS v7.1
002     //hello
003     level 1u
004     ! ' Hello World '
005
006     exit
```

Save the file as "hello.ff" in your ARRIS home directory.
Next you will compile & archive your sigmac to test it out (sorry - next lesson).

Heading/Comments (optional)

```
001     /* hello.ff     sc      03.15.2000      ARRIS v7.1
```

The syntax *slash-asterisk* (**/***) informs the compiler to ignore everything else on this line to the right of the **/***. This is a non-executing *comment* statement and can appear anywhere in your code. Sigmac does not require you to bracket your comments as in C.

You'll appreciate comments much more in a year or two, when you have to modify today's work for tomorrow's ARRIS updates. Even more so, if it is someone else's program.

Command Name

```
002     //hello
```

The syntax - *double slash* (//) specifies the command name for your sigmac. This will be the command name you type in while in ARRIS to execute your sigmac. This should be the first executing statements. The name may contain up to 13 characters, consisting of lowercase "a,b,c, ... x,y,z", numbers "0-9", and the underbar "_" character. It can not begin with a number though.

Tip: A sigmac with the name "ustart" will execute automatically when ARRIS is initiated.

Tip: To maintain sanity, the command name should be reflected in the filename (ff).

Hint #1: Select a 2 or 3 character prefix for your company, being careful not to replicate existing application prefixes on your systems. I use "scg_". This will prevent you from inadvertently reprogramming an ARRIS system command. If you refuse to follow common sense, see hint #2.

Hint #2: While in ARRIS, type the proposed command name you want to use to see if the command name is already in use by an application. If you still refuse to follow common sense, see hint #3.

Hint #3: Stay out of my shop.

Command Level & Classification (optional)

```
003      level 1u
```

The keyword **level** specifies the *priority* and *classification* of your sigmac. The *priority* determines how your sigmac will get in ARRIS when your command is executed from the keyboard. The *level* designation means nothing when your sigmac is executed internally by another sigmac (not entered through the keyboard).

This statement is optional, the default is **1u**.

There are 3 levels of priority:

Level 1 (default) is typically a *draw* or *edit* type of command (sal, box, dla, etc.). Only another level 1 command will terminate it, while allowing levels 2 and 3 to execute their things and return control back to the original level 1 command.

Level 2 is typically a *setup* type of command (pen, color, line type, etc.). Only levels 1 or 2 commands will terminate it, while allowing level 3 to execute its thing and return control back to the original level 2 command.

Level 3 is typically a *toggle* type of command (dsa, xyf, etc.), requiring no user input. Only levels 1 or 2 commands will terminate it, while allowing other level 3 commands to execute their thing and return control back to the original level 3 command.

The *classification* separates your sigmacs from the system's sigmac. The *classification* is specified by appending **u** (default) or **s** to the level #. The **u** refers to *user* and the **s** refers to *system*. This becomes very important when your have a sigmac with the same name as a system command.

Example:

> *If I write a user command called "box" (and I already know there is a system command called "box") and then proceed to type in "box" in ARRIS, my "box" command will execute by default. To force the execution of the system "box" command, I would type in "::box" (preceding it with double colons).*
>
> *Likewise, when using the "box" command in a sigmac, I could type ":box" or "::box". The former, would execute the user "box" command (if found, else it will execute the system "box" command) and the latter will force the execution of the system "box" command (even if there is a user "box" command).*

Hint: Never classify one of your sigmacs as a *system* command. You can really confuse things.

The Body

```
004      ! ' Hello World '
005
006      exit
```

The *flow* is pretty straight forward here, the statements are executed straight down the list. We'll

discuss the many different types of statements used in the body of the sigmac in greater detail in future lessons. Not to leave you hanging . . .

- The exclamation mark (**!**) will echo the following string (enclosed in single quotes) below the prompt line on the ARRIS menu.
- The keyword **exit** will terminate the execution of your sigmac. Not required at the end of this sigmac, for the program will terminate naturally after the last executing statement. I'll put an *exit* statement at the end for code readability (we'll discuss programming style later).
- Blank lines (white space) is ignored by the compiler (meaning you can use blank lines).

Sub-Routines

Although not shown in our first example, *subroutines* follow the last executing statement ('exit' in our example) in the body of your sigmac. We'll discuss subroutines in detail in a later lesson.

```
001     /* hello.ff     sc      03.15.2000      ARRIS v7.1
002     //hello
003     level 1u
004     call print_it
005     exit
006     /*** Subroutines ***
007     print_it::
008             ! ' Hello World '
009     return
```

So now you know the general structure of a sigmac:

- Heading/Comments
- Command Name
- The Body
- Subroutines

Lesson 3
Compiling & Archiving Basics

Re-open the file (hello.ff) from lesson #002 or open any text editor and retype it:

```
001     /* hello.ff     sc      03.15.2000      ARRIS v7.1
002     //hello
003     level 1u
004     ! ' Hello World '
005
006     exit
```

Save the file as "hello.ff" in your ARRIS home directory (if in Windows, see ADE's 'Working Directory' section below for more accurate info).

Compiling

Compiling converts your Sigmac into a faster condensed (00100111010) language (gg). Compiling has a definite speed advantage over interpreted 'macro' languages.

In MS Windows, Sigma's ADE requires some initial setup:

- Go to [File], Select [Set Working Directory. . .]
- Define the *Working Directory* to your ARRIS home directory (or where ever you plan to store your sigmac source files).
- Define the *SM File* to "test.sm" in your home directory. I use 'test.sm' until I get the program working without fatal errors, then I'll archive in a more permanent library. Then I can always delete 'test.sm'.
- Ignore *Global Files* for now.

  Once defined, you won't have to redefine it next time you open ADE, these parameters are saved in the "ade.ini" in the *working directory*.

  Once ADE is initially setup, open the "hello.ff" and hit [c] button (see the use of [rc] below), it will compile.

  Tip: When the ADE compiles, it doesn't compile the code in the ADE's editor window, it compiles the last saved version of it on the harddisk. If you are using the ADE editor to write code, remember to save prior to compiling. The above tip works for you if you use a non-ADE editor for writing. I use Lemmy (a Windows VI editor) to write code, open ADE and then open the same sigmac file in the ADE Editor, minimize it, and compile it over and over again without touching it in the ADE editor. **Caution**: If you inadvertently hit the [Save] in the ADE, overwriting the sigmac file on the hard disk with the ADE's out-of-date version - just go back to Lemmy (or what ever) and resave the current version over top again.

In UNIX, from the command line:

```
% compile  hello.ff          <cr>
```

If successful, it will produce a "hello.gg" file.

We'll discuss compiling options and errors in more detail later on (oh what fun!).

Archiving

Archiving places your sigmac (compiled gg version) into a sigmac library (sm). In ARRIS, you don't load a single sigmac, but rather a library of sigmacs. Two sigmac libraries will load automatically when ARRIS initiates, "file.sm" and "user.sm".

In MS Windows, archiving is handled in the ADE via the [rc] button - it will compile and archive in one step. If you get a page load of compiling errors and they quickly scroll by, you may want to compile only via the [c] button.

In UNIX,

```
% smar  r  test.sm  hello.gg    <cr>
```

where **smar** is the command name and " **r** " is the argument to specify archive, next comes the sigmac library name, and the a list (separated by spaces) of compiled gg files (you may use *.gg).

Load Sigmac Library

Return to ARRIS and type:

```
smf; test.sm    <cr>
```

The ARRIS command **smf** will load a sigmac library

Execute Your Sigmac

In ARRIS, type:

```
hello   <cr>
```

sit back and watch the excitement pile up.

Anytime any modification is made to the source file (ff), you must repeat all these steps for the changes to take affect.

These first 3 lessons should be enough to see something happen, try writing other simple sigmacs - chaining together commands. Next week, "Storing Data Using Variables".

Lesson 4
Storing Data Using Variables

Constants & Variables

Constants are literal values, which are (for sake of a better word) constant. The value can never be changed by the program. By definition, any number or string enclosed in single quotes is a constant and should be evaluated literally.

Example:

> *The number one will always be number one, I can do nothing to change that. The string 'bugs bunny' will always mean "bugs bunny", if I alter this string - it would be a different constant.*

Variables, on the other hand, are labels that represents a *value* and are not to be taken literally. The *value* can be dynamic, in that it can be manipulated by your program.

Variables *names* have a maximum length of 13 characters and must begin with lowercase alpha character. After the first character, alphanumeric and "_" characters may be used. You can not duplicate any Sigmac keyword (`level, global, define, if, elseif, else, endif, repeat, until, loop, while, end, call, goto, exit, return`, etc.) as a variable.

Tip: SDI uses "_" as the 1st character for many of their *global* variable names, so I would stay away from using "_" as the 1st characters- more on *globals* later in this lesson.

Variable Data Types:

- string (ASCII characters)
- integer (whole numbers)
- real (+ fractions)
- point (xyz coordinates)

Variable *declaration*: Sigmac requires that variables must be declared (basically saying "*I am an integer and I demand respect*") to one of the above data types. Luckily, Sigmac uses an *implicit* naming conventions that will automatically declare variables for you. Quite simple to remember, any variable name that begins with:

- **i** is an *integer*
- **s** is a *string*
- **p** is a absolute *point*
- **d** is a relative *point*
- all other alpha characters are *real*.

All variables must be *initiated* prior to use. In other words, you need to know how to assign a *value* to the *variable*. This is done with the *assignment operator* "=" (single equal symbol). When we first assign a variable a value, it is referred to as initializing.

```
width = 1.5
```

The variable is always to the left of the equal symbol and the value (or expression) is always to the right.

Tip: The compiler interprets the value in database unit at the time when your program is executed (may be feet, inches, meters, etc.- who knows). You can force the issue by, `width = 1.5"` or `width = 1.5cm`

Why use variables? The most obvious answer is to store and manipulate data. But even more important is flexible code that is maintainable and readable.

Example:

Let's write a program to draw a 2x4 stud on end. We could use the literal dimensions (1.5" and 3.5") throughout the program.

```
001     //stud2x4
002     level 1u
003
004     ::box; =A(0,0,0); =A(1.5",3.5",0);:
005     ::sal; =A(0,0,0); =A(1.5",3.5",0); =A(1.5",0,0); =A(0,3.5",0);:
006     exit
```

Or we could store these dimensions in variables (width & height) in the beginning of the program and use the variables throughout the program.

```
001     //stud2x4
002     level 1u
003
004     width  = 1.5"
005     height = 3.5"
006
007     plowleft  = A(0,0,0)
008     pupright  = A(width,height,0)
009     pupleft   = A(0,height,0)
010     plowright = A(width,0,0)
011
012     ::box;=plowleft;=pupright;:
013     ::sal;=plowleft;=pupright;=plowright;=pupleft;:
014     exit
```

Then when your boss comes and tells you s/he needs a 2x6 program tomorrow, you can change the "height" variable in one place - done!

```
005     height = 5.5"
```

Rule: Never write it twice.

String Variables

String variables stores one or more ASCII characters. You can declare any variable that does not begin with "s" as a string with:

```
#str bubba
```

When assigning a ASCII string to a string variable, the ASCII character(s) are enclosed in single quotes (always):

```
s2 = 'Mary had a '
s3 = 'little lamb'
s4 = ( s2 + s3 )        /* s4 = 'Mary had a little lamb'
```

Strings are limited to 2048 characters.

Integer Variables

Integer variables stores whole number between -2,147,483,648 and +2,147,483,647. You can declare any variable that does not begin with "i" as an integer with:

```
#int count
```

You can add integer and real variables, but when assigning a real value to an integer variable, rounding is applied accordingly:

```
i2 = 1.5                /* i2 is rounded to 1
i3 = 1.6                /* i3 is rounded to 2
i4 = (5 / 2)            /* i4 is rounds 2.5 to 3
```

Real Variables

Real variables are double precision number between -2,147,483,648.00 and +2,147,483,647.00 with 16 decimal places of accuracy. You can declare any variable as a real with:

```
#real px
```

Point Variables

Point variables stores XYZ coordinates. You can declare any variable that does not begin with "p" as an *absolute* point with:

```
#point origin
```

You can declare any variable that does not begin with "d" as an *relative* point with:

```
#delta origin
```

Each point variable is comprised of 3 (real number) components, X, Y, and Z. To access these XYZ components individually:

```
p2 = (2,3,4)
rx = p2.x               /* rx = 2.0
ry = p2.y               /* ry = 3.0
rz = p2.z               /* rz = 4.0
```

You can add point variables together:

```
p2 = (2,2,0)
p3 = (4,4,0)
p4 = (p2+p3)            /* p4 = (6,6,0)
p5 = (p4/2)             /* p5 = (3,3,0)
p6 = ((p2+p3)/2)        /* p6 = (3,3,0)
/* this last formula calcs the midpoint - cool!
```

When adding point variables with integers or reals, only the X value is modified:

```
p2 = (2,2,0)
p5 = ( p2 + 2.0 )       /* p5 = (4,2,0)
```

When adding point variables with integers or reals, only the X value is modified:

Array Variables

So far the variables we've discuss contain a single value (non-scalar). An *array* (scalar) variable can contain a table of values, similar to a spread sheet. *Arrays* allows you to manage likewise data in an orderly fashion.

Arrays follow the same naming and typing rules (integer, real, string, point) as regular variables. The array's name must be a non-keyword and not previously initialized as a regular variable.

An array can be:

- One Dimensional (list) ilist(0)
- Two Dimensional (table) itable(0,0)
- Three Dimensional (multi-tables) itables(0,0,0)

Each array consists of a variable *name* and *subscripts*. Each *subscript* is a positive integer (including 0) within the "( )" and separated by commas. The 3 subscripts formulate an address to a cell, where the data is stored. You can not mix data types within the array's cells.

Variable's Life Span

Typically a variable (any data type) has a life span for the duration of the executing sigmac. Once the sigmac terminates, so does its variables.

Introducing *global variables*. *Global* variables have a life span of ARRIS itself, only when you exit ARRIS are global variables terminated. To declare a variable as global:

```
#global  porigin,  pmin,  pmax  /* Mult vars are sep by commas.
```

If you need to declare a variable as global and change its data type:

```
#global  #int  scg_counter
```

Therefore global variables can be utilized by any sigmac as long as each sigmac has the above global declaration or you can manage all global variables in a single file to be included at the time of compilation. More on this concept when we discuss compiling in depth.

Global variables can either be an array (scalar) or non-scalar variables. Global variables follow the same naming rules as regular variables.

You can verify that a global variable is not used by another application by echoing it in ARRIS with the application loaded.

```
!scg_counter
```

More than you want to know:

There are system contants (e.g. #true = 1, #false = 0, #vinf = 2147483647, etc.). The value of a system constant can be obtained by echoing it in ARRIS:

```
!#vinf      /* largest possible number in ARRIS
```

There are system variables (e.g. #fcol = current color setting in ARRIS or #fpen = current pen setting in ARRIS). The value of a system variable can be obtained by using $getvar() or $getflg() and echoing it in ARRIS:

```
!$getvar(#vlastkey)     /* ASCII value for the last key entered
```

or

```
!$getflg(#flin)         /* returns the current line type
```

There is not a good way to distinguish a system #variable from a system #constant, they both are prefixed with #. Most #variables beginning with "#f" are object (entity variables) flags.

System global variables are not to be confused with "*system #variables*". *System #variables* are declared in the private C-level of ARRIS, while the *system global variables* are declared in the public sigmac level. SDI manages their global variables in files ending in "__glob.ff" in their source code directories.

Lesson 5
Need Input!

We now have *variables* to put stuff in. In our 2x4 stud program, it will draw a 2x4 stud correctly every single time. Only problem is, it draws the stud at absolute zero (line #007) every single time. We call this *hardcoding*, where there is no flexibility.

```
001     //stud
002     level 1u
003
004     width  = 1.5"
005     height = 3.5"
006
007     plowleft  = A(0,0,0)
008     pupright  = A(width,height,0)
009     pupleft   = A(0,height,0)
010     plowright = A(width,0,0)
011
012     ::pen;=2
013     ::box;=plowleft;=pupright;:
014     ::pen;=1
015     ::sal;=plowleft;=pupright;=plowright;=pupleft;:
016     exit
```

Lines #012-015 call other sigmacs (future lesson). Everything else, we have discussed and you should be well versed on.

User Input

We need a way to add flexibility - to conform this sterile program in to something useful. Lets begin by changing line #007 to accept input from the user:

```
007     plowleft = $inp(#point,'Lower Left Corner of Stud')
```

The $utility, `$inp()` pauses the program to retrieve input from the keyboard or mouse (e.g. user).

- The first argument within the parenthesis is the *data type* (see Lesson #003) of the variable to the left side of the *assignment operator* (=). In our example, we are looking for a point value returned by the user.
- The second argument is a string, which will be displayed on the ARRIS prompt line.
- There are more powerful options for `$inp()`, but these are the basics.

Now that our start point could be anywhere, we need to adjust the 3 remaining corner points to be *relative* to the new start point. We can do this by adding the 3 corner points to the start point with the width and height dimensions. Rewrite lines #008-010 as follows:

```
008     pupright  = A(plowleft + (width,height,0))
009     pupleft   = A(plowleft + (0,height,0))
010     plowright = A(plowleft + (width,0,0))
```

? Input

When calling another Sigmac, you must provide all expected input, therefore the following will execute the "pen" command, ask for the "Logical Pen", and not return control to your sigmac to finish execution of lines #013-016.

```
012     ::pen
```

To maintain control, we can either supply the input expected via a variable or constant (as in our original sigmac at the top of this lesson) or use a *question mark* (`?`) as a place holder. This will pause the execution of your program and prompt the user for input utilizing the 'called' sigmac's prompt string. Once the user answers the prompt, the execution control will return to your sigmac. To illustrate, we could rewrite lines #012 (and #014) to:

```
012     ::pen;?
```

Hint: Notice no equal symbol (=) used with the "?", as in assigning a variable. The down side is that this method does not offer you a chance to perform error checking (. You have to depend on the sigmac to perform adequate error checking.

More Input

Our sigmac would be even more useful with even less *hardcoding*. Let's do some redesigning and make our program draw more 2x studs.

```
001     //stud
002     level 1u
003
004     width  = 1.5"
005
006
007     stud  = $inpc(#strn,'Draw Which Stud',1,'2x4','2x6','2x8')
008     if stud == '2x4'
009             height = 3.5"
010     elseif stud == '2x6'
011             height = 5.5"
012     elseif stud == '2x8'
013             height = 7.5"
014     endif
015
016     plowleft  = $inp(#point,'Lower Left Corner of Stud')
017     pupright  = A(plowleft + (width,height,0))
018     pupleft   = A(plowleft + (0,height,0))
019     plowright = A(plowleft + (width,0,0))
020
021     ::pen;?
022     ::box;=plowleft;=pupright;:
023     ::pen;?
024     ::sal;=plowleft;=pupright;=plowright;=pupleft;:
025     exit
```

Line #007 adds flexibility by introducing `$inpc()`, which takes the `$inp()` to the next level by adding choices to select from in a pull-up menu on the prompt line.

- The first 2 arguments are similar to `$inp()`.
- The third argument specifies whether the user is limited to the list of choices or not. "1" limits the user to the options in the list, whereas "2" or "3" does not.
- The remaining arguments is the list of choices separated by commas.

Lines #008-014 processes the user's choice via `if/elseif/endif` conditional statements (future lesson).

Displaying Additional Information

As discussed earlier in our "hello.ff" sigmac, we can use the *exclamation symbol* (!) to display a string below the ARRIS prompt line. Let's change line #015 in the above example to:

```
015     !'Drawing A '+stud+' Stud'
016     plowleft  = $inp(#point,'Lower Left Corner of Stud')
```

This allows us to give the user more information when answering your prompts. Other $utility options that display additional information include `$prmess()`, `$error()`, `$ierror()`, and `$ferror()`. These will be discussed in a later lesson.

Tip: Once we open out sterile program to the outside world, we invite errors entered from the outside world (eg users, text files, etc.). Imediately following any input statements, you should follow with `if/else/endif` statements to verify the data entered by the user is within the valid range of values expected by your program. More on *error checking* later when discuss *conditional statements*.

Lesson 6
Calling All Sigmacs

In lesson #005 (*Need More Input!*), we had a 2x4 stud program that called other sigmacs. This weeks lesson will explore this subject in more detail.

```
001     //stud
002     level 1u
003
004     width  = 1.5"
005
006
007     stud  = $inpc(#strn,'Draw Which Stud',1,'2x4','2x6','2x8')
008     if stud == '2x4'
009             height = 3.5"
010     elseif stud == '2x6'
011             height = 5.5"
012     elseif stud == '2x8'
013             height = 7.5"
014     endif
015
016     plowleft  = $inp(#point,'Lower Left Corner of Stud')
017     pupright  = A(plowleft + (width,height,0))
018     pupleft   = A(plowleft + (0,height,0))
019     plowright = A(plowleft + (width,0,0))
020
021     ::pen;?
022     ::box;=plowleft;=pupright;:
023     ::pen;=2
024     ::sal;=plowleft;=pupright;=plowright;=pupleft;:
025     exit
```

Lines #021-024 call other sigmacs.

Classification

Way back in lesson #002 (*First Sigmac "Hello World"*), we spoke about a Sigmac's *classification* and level. You may want to review lesson #002 before reading on. To quickly summarize:

The 'double colons' forces ARRIS to execute the '*system*' command??:

```
099     ::sal
```

Whereas the 'single colon' will execute the '*user*' command (if any), else it will fall back and execute the 'system' command by the same name.

```
099     :sal
```

Easy enough! But if you stop reading now, your sigmac will pass control to the `sal` command and terminate. Never returning the control to your sigmac and execute the remaining code after `sal` command. 'Control' is the current focus of ARRIS.

The sigmac's 'level', as specified in the `level` statement, has nothing to do with the 'control' being passed between sigmacs or mnemonics.

Control

Get Control! You have to first figure out what `sal` expects in way of prompts/input. So first, manually execute `sal` in ARRIS and write down the prompts and user input expected.

Now change the line to read:

```
099     ::sal;?;?;?;?;?;:
```

- The `?` is a place holder for each expected prompt/input (*lesson #005 - Need Moe Input*).
- The `;:` (semi-colon/colon) at the end will terminate a repeating command.

Both of these **WILL** ensure the return of the control to your sigmac and finish executing everything that follows.

Note: You could use `?u` to force the input to come from the keyboard and NOT another sigmac or mnemonic command. I wouldn't, it gives your sigmac an attitude and renders it relatively useless when called by another sigmac.

Customize

You may want to change things up a bit with a more pertinent prompt relative to your sigmac.

```
097     p1=$inp(#point, 'First Fence Point to Define Search Window')
098     p2=$inp(#point, 'Second Fence Point to Define Search Window')
099     ::cpa;=p1;=p2;?;?;:
```

Special Key Options

But the `cpa` command has an optional F10 (function key) in the first prompt to define special fence options (all, complex, workplane, etc.). We could try . . .

```
099     ::cpa;='F10';?;?;?;:
```

I think not - it couldn't be that easy. Introducing the `@` character for specifying 'point' input via function keys. The left hand side of the `@` specifies a point variable and the right hand side specifies the actual function key (1-10, but 10 is represented by 0) to be applied to said point variable.

Since we don't require an actual point variable in our `cpa` example, we'll use "0" as a place holder for the point variable. Since we want to specify the F10 key, we'll use `@0`. Therefore . . .

```
099     ::cpa;=0@0;='all';?;?;:
```

Will execute the `cpa` command, implement the optional F10 list of specialize fences, select "all" from the list, and continue with the standard 'reference point' and 'new point location' prompts.

Another cool feature of `@`, is that it can be used as a crude 'entity select'. For example:

```
098     p1=$inp(#point, 'Select Object')
099     :dosomething; =p1
```

by itself will not position the database pointer to any object in the drawing, because the typical F1 input will only return the point coordinates under the cursor - not the point coordinates of the desired object. It will work if the user knows to use the F3 when selecting the object. Why not make it easy for the users by specifying the F3 key within your sigmac.

```
098     p1=$inp(#point, 'Select Object')@3
099     :dosomething; =p1
```

or

```
098     p1=$inp(#point, 'Select Object')
099     :dosomething; =p1@3
```

By appending `@3` at the end, the database pointer IS positioned at the desired object - ready for action by your sigmac. Granted, there are better ways of positioning the database pointer.

Likewise:

```
099     p1=$inp(#point, 'Select Point')@2      /* F2 to override XY forcing
099     p1=$inp(#point, 'Select Line')@5
099     p1=$inp(#point, 'Select Text')@6
099     p1=$inp(#point, 'Select Intersection')@7
099     p1=$inp(#point, 'Select RI')@8
```

Lesson 7
Evaluating Data

It is now time to compare *values* against one another. *Values* can come in the form of *variables* or *constants* (Lesson #004), numerals or character strings.

The following *operators* are used in conjunction with *conditional statements* (`if/endif`, etc.) to control the flow or logic of your program. Bare with me, it tough to talk about *evaluating data* without getting wrapped up in *conditional statements*. To put both in this weekly lesson would violate the main principle of keeping the lessons as simple as possible. Next week we'll jump into *Conditional Statements*.

Expressions

Expressions can be either be a single value, an equation that equates to a single value, or a $utility that returns a single value. Again, the value can be either a number or string.

Relative Comparisons

Relational operators require 2 values, making them *binary* operators. Relational operators numerically (which includes ASCII values) compares one expression relative to another expression for trueness.

Relational operators test for:	2 Syntax Options	
• Equality	= =	.eq.
• non-equality	<>	.ne.
• less than	<	.lt.
• greater than	>	.gt.
• less than or equal	<=	.le.
• greater than or equal	>=	.ge.

You can use either syntax options mentioned above, Sigmac recognizes both.

Note: The equal (=) character is used in both testing for equality and assigning a value to a variable. The former uses a double (==) equal characters and the latter uses a single (=) equal character.

Relational Operator Examples:

```
006     i1 = 100
007     i2 = 100
008     i3 = 200
009
010     if i1 == i2             /* Test for Equality
011             . . . . .
012     endif
013     if i1 <> i3             /* Test for Non-Equality
014             . . . . .
015     endif
016     if i1 <= i2             /* Test for Less-Than or Equal
```

```
017             . . . . .
018     endif
019     if i3 >= i2             /* Test for Greater Than or Equal
020             . . . . .
021     endif
022     if i1 < i3              /* Test for Less Than
023             . . . . .
024     endif
025     if i3 > i1              /* Test for Greater-Than
026             . . . . .
027     endif
```

All *relational* comparisons in the above examples evaluate to #true, therefore the subcode within the `if/endif` statements will execute. Get the idea. Evaluating data in combination with conditional statements will control the flow of your program. In other words, which part of your program will execute under which conditions.

Logical Comparisons

Logical operators compare (not numerically, but logically) one expression's *trueness* relative to another expression's *trueness*, returning either #true (1) or #false (0).

Logical operators test for:	Syntax
• Both equations are #true	.and.
• Either/Both equations are #true	.or.
• Either/Or equations is #true	.xor.
• Negation	.not.

Most logical operators require 2 expressions, making them *binary* operators. Only the 'negation' (.not.) operator requires a single expression, making it a *unary* operator.

Logical Operator Examples:

```
006     i1 = 100
007     i2 = 100
008     i3 = 200
009
010     if i1 == i2 .and. i1 < i3       /* Both Expressions Must Be True
011             . . . . .
012     endif
013     if i1 <> i3 .or.  i1 == i2      /* Either or Both May Be True
014             . . . . .
015     endif
016     if i1 <= i2 .xor. i1 == i3      /* Either, But Not Both May Be True
017             . . . . .
018     endif
019     if .not.(i3 == i2)                      /* Test for Non-Equality
020             . . . . .
021     endif
```

All *logical* comparisons in the above example evaluate to #true, therefore the subcode within the `if/endif` statements will execute. Line #019 returns #false within the parenthesis and the negation (.not.) of `#false` is `#true`. Lines #010, 013, and 016 are also known as *compounded*

statements, evaluating multiple expressions.

We could have written line #010 as:

```
010     if i1 == i2
011             if i1 < i3
012                     . . . . .
013             endif
014     endif
```

We could have written line #013 as:

```
013     if i1 <> i3
014             . . . . .
015     endif
016     if i1 == i2
017             . . . . .
018     endif
```

Line #016 would be more difficult to write without logical operator, `.xor.`.

We could have written line #019 as:

```
019     if i3 <> i2
020             . . . . .
021     endif
```

Lesson 8
Conditional Statements

Last lesson (#007 - Evaluating Data), we discussed how to compare expressions for trueness. *Conditional statements* allow you to design program flow or logic into your program. By evaluating certain conditions, you can dictate which subcode sets to execute, how many times, and when.

All conditional statements begin and end with Sigmac keywords. All keywords are lowercase.

If/Endif

The if/endif conditional statement is a switch type statement - if #true, execute the subcode - if not, skip it.

Example #1 - Single Switch:
```
006     if i1 == 100
007             /* subcode . . . .
008     endif
```

Example #2 - Multiple Switch Option:
```
006     if i1 <= 100
007             /* subcode . . . .
008     elseif i1 <= 200
009             /* subcode . . . .
010     elseif i1 <= 300
011             /* subcode . . . .
012     endif
```

You can place as many `elseif` statements as required. ARRIS will evaluate these conditional statements in the same order as you write them. This is important because once any `if/elseif` statement is evaluated to be `#true`, the appropriate subcode is executed. The program's control is directed to the matching `endif` statement and the remaining `elseif` statements are never evaluated nor executed.

Tip: On multiple switches, it may be desirable to order evaluations from specific conditions to general condidtions.

Example #3 - Adding a Default Action
```
006     if i1 <= 100
007             /* subcode . . . .
008     elseif i1 <= 200
009             /* subcode . . . .
010     else
011             /* subcode . . . .
012     endif
```

If `i1` is greater than `200`, the subcode listed on line #011 will execute by default. You can only place one `else` statement after all `if/elseif` statements and prior to the `endif` statement.

Loopwhile/Endloop

The `loopwhile/endloop` set is a program loop with the condition evaluated at the beginning. This means the subcode may never be executed (main difference with repeat/until).

Once the stated condition returns #true, the loop will release the program's control.

Example #4 - Simple Loop

```
006     loopwhile (ik < 10)
007             /* subcode . . . .
008     endloop
```

Assuming `ik` is being incremented in the subcode, this loop will execute the same subcode x number of times and exit at the `endloop` statement (#008).

Example #5 - Add a Loop Counter

```
006     loop ik=0 while ik < 10
007             /* subcode . . . .
008     end ik=(ik+1) loop
```

Now `ik` is being initiated in the `loopwhile` statement (#006) to zero and is being incremented (or possibly decremented) in the `endloop` statement (#008), this loop will execute the same subcode 10 times and exit at the `endloop` statement.

Repeat/Until

The `repeat/until` set is a program loop with the condition evaluated at the `end`. This means the subcode will always be executed at least once (main difference with `loopwhile/endloop`).

Example #6 - Simple Loop

```
006     repeat
007             /* subcode . . . .
008     until ik == 10
```

Once the stated condition returns `#false`, the loop will release the program's control. Assuming `ik` is being incremented in the subcode, this loop will execute the same subcode x number of times and exit at the `until` statement (#008).

There is no builtin *initialization* or *incrementor/decrementor* in the `repeat/until` statements.

Nesting

Nesting is when you place one of these statement groups within another. You can place a `if/endif` inside a `repeat/until` loop or vice-versa. Each statement group within another is considered a *level*. You may nest as many levels as required by your program.

Example of Nesting (3 levels):

```
006     if ik < 10
007             repeat
008                     if ik > 5
```

```
009                             /* subcode . . . .
010                     endif
011                     ik=(ik+1)
012             until ik == 10
013             . . . .
014     endif
```

The only rule concerning nesting is that each *level* terminates before its parent level terminates.

Example of **Illegal** Nesting:

```
006     repeat
007             if ik == 10
008     until ik == 100
009             endif
```

Lesson 9
Conditional Statements

Last lesson (#008 - Conditional Program Flow), we discussed conditional statements and their affect on the program's flow by first evaluating expressions. UnConditional statements allows you to direct the program's flow to another part of your program without evaluating conditions.

Most unconditional statements requires a programmer defined keyword called a label. A label is like a bookmark in your program. A label is identified by either a 1 or 2 colons (:) suffixed.

Note: Do not confuse these colons used as suffixes with the colons used as prefixes when calling sigmac programs (Lesson #006 - Calling All Sigmacs).

Goto

The goto statement redirects the program's flow to the specified label (suffixed by a single colon) without any expectation of returning to the goto statement.

Example #1:
```
006     goto skip
007     . . . .
        . . . .
022     . . . .
023     skip:
024     . . . .
```

Lines #007-022 will never execute. The goto statement on line #006 redirects the program's flow to line #023.

Call/Subroutine/Return

The `call` statement redirects the program's flow to a subroutine. A subroutine is a subset of code, often called more than once by the main body of the program (one subroutine may be called from multiple `call` statements). All subroutines reside at the end of the written program, after the last executing statement in the main body. The order of individual subroutines do not matter.

The power of subroutines is in "writing the code once". This reduces logic errors, increases maintainability and is easier to debug.

Example #2 - Call Subroutine
```
004     . . . .
005     isq=4
006     call calc_sq
007     /* isq now equals 16
008     . . . .
        . . . .
021     . . . .
022     exit
023     /** Subroutines begin here **
```

```
024     cal_sq::
025             isq = (isq * isq)
026     return
```

Line #006 calls `calc_sq::` on line #024. The subroutine executes its subcode. The return on line #026 will return the program's control to line #007. Now lines #007 thru #022 will execute.

A subroutine begins with a label suffixed by double colons and ends with the `return` statement. The `return` statement redirects the program's flow back to the statement following (#007) the originating `call` statement, making `call/return` *bi-directional*.

A subroutine may have more than one `return` statement, or the subroutine may call other subroutines, or the program may terminate in a subroutine.

Example #3 - Call Subroutine with several exit points

```
004     . . . .
005     isq=4
006     call calc_sq
007     /* isq now equals 16
008     . . . .
        . . . .
021     . . . .
022     exit
023     /** Subroutines begin here **
024     cal_sq::
025             if isq == 0
026                     exit
027             elseif isq < 0
028                     isq = 0
029                     $error(0,'Negative Numbers')
030                     return
031             else
032                     isq = (isq * isq)
033             endif
034     return
```

Line #006 calls `calc_sq::` on line #024. The subroutine executes its subcode. Either return on line #030 or 034 will return the program's control to line #007. Now lines #007 thru #022 will execute.

Exit

The exit statement terminates the program's flow cleanly. `$ferror()` will abort your program with a fatal error message.

Lesson 10
Manipulating Numbers

We've discussed the benefits of using variables in lesson #004, lower maintenance and flexibility. Today we're going to expand on the flexibility aspect of using variables.

There are 3 basic types of data; numbers, text, and points. The next 3 lessons will discuss each one of these individually. *Numbers* include both real and integer data types.

Operators

The basic operators include multiplication (*****), division (**/**), addition (**+**), and subtraction (**-**). A higher level of operators include exponential notation (****** or ^) and scientific notation (**e**).

```
005     i = 6+2  /* i=008
006     i = 6-2  /* i=004
007     i = 6*2  /* i=012
008     i = 6/2  /* i=003
009     i = 6**2 /* i=036
010     i = 6^2  /* i=036
011     i = 6e2  /* i=600
```

Evaluation Order

ARRIS follows the typical evaluation priorities:

- 1st) Exponential or Scientific Notations
- 2nd) Multiplication or Division Operators
- 3rd) Addition or Subtraction Operators

```
012     i = 6^2+3*4
```

If you did not have priority rules, you could evaluate line #012 as (((6^2)+3)*4)=156 or (6^(2+(3*4)))=78364164096 or ((6^2)+(3*4))=048, the latter is true with the above evaluation order when parenthesis are not applied.

```
013     i = 6+2-3+4
```

The same is true with line #013, (((6+2)-3)+4)=009 or ((6+2)-(3+4))=020, the former is true when parenthesis are not applied. Why? Because if different operators have the same priority level, they are evaluated from left to right. As you can see in the above examples, parenthesis have the highest priority level and can change the evaluation order.

Parenthesis can be nested. The inner-most set of parenthesis are evaluated first, working its way to the outer-most set of parenthesis. When 2 or more sets of parenthesis fall on the same level, they are evaluated from left to right.

Tip: Not only should parenthesis be used to dictate the evaluation order, they should also be used to increase the readability of your equation.

Shortcut Operators

Sigmac offers a couple of shortcuts to increment or decrement a variable by one.

```
014     ivar++  /* same as ivar=(ivar+1)
015     ivar--  /* same as ivar=(ivar-1)
```

Dot Operators

Dot operators perform specialize functions (note data types required and returned):

integer = integer1 **.mod.** integer2

Returns an integer of the remainder after integer2 is divided into integer1.

```
016     i = 12.mod.2    /* i=2
017     ieven = ieven+(ieven.mod.2)
018     /* Will always round i2 to the next higher even number
```

real = real1 **.rnd.** real2

Rounds a real1 to the nearest multiple of real2. Rounding differences greater than or equal to 0.50% of r2 will round up.

```
018     r = 12.50.rnd.3.0       /* r=12.0
019     r = 13.50.rnd.3.0       /* r=15.0
```

real = real1 **.min.** real2 **&** real = real1 **.max.** real2

Returns the smaller or larger of the 2 numbers (real1 & real2).

```
020     r = 12.50.min.3.0       /* r=03.0
021     r = 12.50.max.3.0       /* r=12.5
```

$utilities

There are numerous $utilities that deal with algebra and trigonometry. I'll just quickly list them here, leaving the exploration to you and a much later lesson.

Algebra	Returns:
$abs(r1)	Absolute value of r1
$frac(r1)	Fractional portion of r1
$int(r1)	Whole number portion of r1 (always rounds down)
$sign(r1)	Either -1 or +1, depending on whether r1 is negative or positive
$sqrt(r1)	Square root of r1

Trigonometry	Returns:
$sin(r1)	Sine of angle r1
$asin(r1)	Arcsine of r1
$sinh(r1)	Hyperbolic sine of r1
$cos(r1)	Cosine of angle r1
$acos(r1)	Arcosine of r1
$cosh(r1)	Hyperbolic cosine of r1

$tan(r1)	Tangent of angle r1
$atan(r1)	Arctangent of r1
$tanh(r1)	Hyperbolic tangent of r1

Lesson 11
Manipulating Text

A single piece of *text* is referred to as a string in the wide world of programming. A *string* consists of 1 or more characters from the ASCII table. The *ASCII* table assigns an integer (0-255) to each letter, number, and punctuation we see on the screen. Why? Because the computer only understands numbers, in fact only zeros and ones - but we won't go there.

Comparisons

The thing about strings are, when you evaluate them - you are evaluating its underlying ASCII value. For example, we can write:

```
005     if 'a' < 'd'
```

Since the ASCII value of "a" is 97 and "d" is 100, the `if` statement returns true. Not knowing this, you may have written:

```
006     if $asc('a') < $asc('d')
```

Which is perfectly ok, just long winded.

```
007     if 'abc' == 'def'
```

Line #007 - on the first pass, the ASCII value of "a" is compared with the ASCII value of "d". It does not equal, therefore the `if` statement is false and "b" and "c" are never evaluated with "e" and "f".

```
008     if 'ab' < 'af'
```

Line #008 - on the 1st pass, "a" == "a", but on the 2nd pass, the ASCII value of "b" is compared with the ASCII value of "f". The ASCII value of "b" is less than "f", therefore the `if` statement is #true.

Operators

The only basic operator is addition (+) or concatenation.

```
009     s = 'ARRIS'+'cad'  /* s='ARRIScad'
010     s = 'Sigma'+'Design'+'International'
```

Line #010 will return "SigmaDesignInternational" - Don't forget spaces!

Dot Operators

Dot operators perform specialize functions. Character positioning is base1, the first character in the string is position 1.

string = string1 **.left.** iposition

Returns the left-part of string1, up to and including the (iposition)th character.

```
016     s = 'ARRIScad'.left.5   /* s='ARRIS'
```

string = string1 **.right.** iposition

Returns the (iposition) right-most characters in the string1.

```
017     s = 'ARRIScad'.right.3  /* s='cad'
```

string = string1 **.from.** iposition

Returns all characters to the right of (& including) the (iposition)th character in the string1.

```
018     s = 'ARRIScad'.from.6   /* s='cad'
```

string = string1 **.pick.** iposition

Returns the (iposition)th character in the string1.

```
019     s = 'ARRIScad'.pick.6   /* s='c'
```

string = string1 **.pos.** string2

Returns the beginning position of the first left-most occurrence of string2 in string1.

```
020     i = 'ARRIScad'.pos.'R'  /* i=2
021     i = 'ARRIScad'.pos.'RR' /* i=2
```

string = string1 **.rpos.** string2

Returns the beginning position of the first right-most occurrence of string2 in string1.

```
022     i = 'ARRIScad'.rpos.'R'         /* i=3
023     i = 'ARRIScad'.rpos.'RRI'       /* i=2
```

$utilities

There are numerous $utilities that deal with strings. I'll just quickly list them here, leaving the exploration to you and a much later lesson.

	Returns:
$acs(s1)	ASCII value
$chk(s1)	True if s1 is numeric
$chr(i1)	Character for i1 (ASCII value)
$len(s1)	Number of characters in s1
$str(r1)	Converts a number to a numeric string (r1 to s1)
$trim(s1)	Trims leading and trailing spaces
$val(s1)	Converts a numeric string to a number (s1 to r1)

Lesson 12
Manipulating Points

This subject has more depth than what I present here as a basic intro. I welcome others to comment and add their points of view.

Point data has a different data structure. Each DB point has 3 components; X, Y, and Z distances from absolute zero (A(0,0,0)). Each one of these *components* can be individually accessed by:

```
005     p1 = (2,4,6)
006     rx = p1.x  /* =2
007     ry = p1.y  /* =4
008     rz = p1.z  /* =6
```

A pixel point will only have the X and Y components. Additionally, the data structure of a point has an index value. Some $utilities will set this *index value* to #none, depending on the success or failure of the $utility. This index value can be accessed by:

```
009     p1 = $fndpt(p2)
010     if p1.i == #none
011             $ferror(0,'Point Not Found')
012     endif
```

Vectors

Every point defines a vector from A(0,0,0). Each *point vector* has a length, a rotation angle about the Z-axis, and an elevation angle above/below the work plane. A unit vector is identical, but always has a length of one unit (foot? meter? who cares? - they are used to simplify the process of calculation angles and elevations).

Operators

The basic operators include multiplication (*****), division (/), addition (+), and subtraction (-) with the following rules:

- A point can not be multiplied or divided by another point.
- A point can be multiplied or divided by a number (real or integer).
- A point can be added to or subtracted from another point.
- When adding or subtracting numbers (real or integer) to a point, you must specify it as an XYZ coordinates in order to affect the Y or Z components.

```
013     p1 = (2,4,6)
014     p2 = (1,3,3)
015     p  = p1+p2              /* = (3,7,9)
016     p  = p1-p2              /* = (1,1,3)
017     p  = p1*2               /* = (4,8,12)
018     p  = p1/2               /* = (1,2,3)
019     p  = p1+7               /* = (9,4,6)
020     p  = p1+(7,5,0) /* = (9,9,6)
```

Tip: By adding 2 points and dividing the resulting point in half - you have determined the mid point between the original 2 points (pmid = ((p1+p2)/2)).

Evaluation Order

Same as in lesson #010 - Manipulating Numbers.

Dot Operators

Dot operators perform specialize functions(degrees is a real number value):

point = point1 **.rtx.** degrees **&** point = point1 **.rty.** degrees **&** point = point1 **.rtz.** degrees

Rotates the point around an axis the specified number of degrees.

```
021     p = A(1,0,0).rtz.90 /* p=A(0,1,0)
022     p = A(1,0,0).rty.90 /* p=A(0,0,1)
```

$utilities

There are numerous $utilities that deal with points. I'll just quickly list them here, leaving the exploration to you and a much later lesson.

	Returns:
$cenpt()	Center point based on the 3 specified points
$colin()	Determines if the 3 specified points are colinear
$hand()	Determines for p3 - which side of the line p1-p2 it falls on
$len()	3D length from p1 to A0
$len2()	2D length from p1 to A0
$lint()	Intersection data about 2 specified line segments
$perp()	Perpendicular foot from p1 on line p2-p3

Contact Steve Clark at: clarks@mindspring.com
or 011-506-271-0241 (voice & fax)
or SJO 1219, PO Box 025216, Miami, FL. 33102-5216