Introduction to the KEK SADScript Environment: Adding Custom Functions

Christopher K. Allen
KEK, Japan

This exposition is currently incomplete. However, it does include some useful information on creating custom functions for the SADScript environment. The major shortcoming here is the lack of information on passing matrix arguments to the compiled function. I was unable to identify this procedure. Perhaps in the future someone more familiar with the SAD environment can update this note, augmenting the areas which I have failed to include.

1 Introduction

In general, to add your own functions to SADScript there are several procedures you must follow. Once completed, however, this function is available to all users of SADScript. In this short note we follow the example of adding a function called `MyFunct` to the SADScript environment. The actual computations done in this function will be contained in a subroutine `MyFunctSub` in the file `MyFunctSub.f`.

1.1 Function Registration

You must register your SADScript function with the interpreter. This can be done in the subroutine `tfinit` in the file `tfinitn.f`. Within these functions you must make a call to the subroutine `itfunaloc`. An example registration is listed in Excerpt 1. The arguments of `itfunaloc` are listed in Table 1.

:

```
map(1) = 0
map(2) = 0
ieval(1) = 0
ieval(2) = 0
i=itfunaloc('MyFunct', 1037, 2, map, ieval, 0);
```

Excerpt 1: example function registration

Argument	**Type**	**Description**
1st (e.g., 'MyFunct')	String	SADScript function name
2nd (e.g., 1037)	Integer	SADScript registration identifier (must be unique). Index on function lookup table.
3rd (e.g., 2)	Integer	Number of function arguments
4th (e.g., map)	Integer Array (1 elem/arg)	SADScript expansion flag 0 – Any listed arguments are passed to the function as a structure on the SAD stack frame 1 – SADScript interpreter expands listed arguments into separate function calls, one call for each element in the list
5th (e.g., ieval)	Integer Array (1 elem/arg)	Mathematica-type evaluation scheme? 0 – Early evaluation (normal) 1 – Delayed evaluate non-mathematical function 2 – Delayed evaluate mathematical function.

6th (e.g., 0)	Integer	Function Type 0 – normal function 1 – special function 2 – special function

Table 1: `itfunaloc` argument descriptions

1.2 Function Binding

SADScript functions are bound at runtime using their registration ID, or regId. There is a set of subroutines that perform this action based on the numeric range of the regId.

regId Range	File
220 to 2000	tfefun1
2000 to 3000	tfefun2
3000 to 4000	tfefun3

The binding is implemented with a `FORTRAN` indexed `GOTO` statement in the subroutine `tfefun?`. Therefore, according to your `regId` value, you must add a line label to the index table and place your function call at the indicated line in one of these files. An example of this action is shown in Excerpt 2 for the original example started in Excerpt 1. This addition was added to subroutine tfefun1 since the regId was 1037. Within the subroutine `tfefun1` there is a line `id = id0 – 1000` which creates the table index `id` seen in Excerpt 2. Since regId = 1037, the value of `id` is 37, the 37th label on the table is 5370. Thus, the binding eventually goes to label 5730 in the subroutine `tfefun1` where you add a call to your custom SADScript function.

```
      go to (5010,5020,5030,5040,5050,5060,5070,5080,5090,5100,
     $       5110,5120,5130,5140,5150,5160,5170,5180,5190,5200,
     $       5210,5220,5230,5240,5250,5260,5270,5280,5290,5300,
     $       5310,5320,5330,5340,5350,5360,5370
     $     ),id

c          ELEM TWIS LINE CalE TraP CalO DyAp RspM Mast FLAG
c          Mcad exDA InDA MAP  FFS  RadF RadS Flag ExBL BLNm
c          SetE PyAg TclA Tcl  ClLi ExpB GetM ClTr LiTr RGBC
c          CPro TcA1 CaSy TkOA CaSD TcSR MyFn
```

Excerpt 2: adding the new SADScript function binding

The call to your custom function should appear something like that seen in Excerpt 3. At the line label 5370 you make the call to your subroutine `MyFunctSub`, where the actual computation is implemented, then return from the subroutine `tfefun1`. Note that there are special arguments to your subroutine `MyFunctSub`. These are the arguments passed to `tfefun1` from the SADScript environment as a result of a call to your custom function `MyFunct`. These arguments must be unpacked to get the variables actually intended for the new SADScript function.

```
5370 call MyFunctSub(isp1, itx, iax, vx, irtc)
     return
```

Excerpt 3: call to custom function subroutine

1.3 SADScript Argument Packing

The SADScript environment passes a set of standard arguments to any function during its invocation. An example of this is shown in Excerpt 4 where we have the FORTRAN declaration of our custom subroutine `MyFunctSub`. As you can see, we have five arguments `isp1`, `itx`, `iax`, `vx`, and `irtc`. The values of these arguments are shown in Table 2.

```
      SUBROUTINE MyFunctSub(isp1, itx, iax, vx, irtc)

      IMPLICIT NONE
      INCLUDE 'inc/TFMACRO1.inc'  ! SAD global simulation variables
      INCLUDE 'inc/TFSTK.inc'     ! SAD global stack variables

C     ARGUMENTS - The interpretations are mostly educated guesses
      Integer*4   isp1    ! stack offset pointer
      Integer*4   itx
      Integer*4   iax
      Integer*4   irtc    ! return code
      Real*8      vx      ! returned value for single value function?
```

Excerpt 4: SADScript standard arguments

Argument	Type	Description
`Isp1`	Integer*4	SAD stack frame pointer. This is the stack base pointer, so subroutine arguments are contained above this pointer location. The global variable `isp` has the location of the top of the stack.
`Itx`	Integer*4	Return value type. SAD has a set of enumerations in `TFCODE.inc` specifying the return type. ntfoper (=0) ntfreal (=1) scalar of type REAL*8 ntflist (=3) a list of REAL*4 ntflistr (=4) a list of type REAL*8 ntfstkseq (=5) ntfstk (=6) ntfstring (=101) a string ntfsymbol (=201) ntfpat (=203) ntfarg (=205) ntffun (=ntfoper) ntfdef (=ntfsymbol)
`Iax`	Integer*4	Pointer to the returned value body. The argument `iax` points to a data structure for passing function data. When the function returns a real scalar value then `iax` should be zero.
`irtc`	Integer*4	The return status code 0 – function call was successful 1 – error occurred in function call
`Vx`	Real*8	Convenience argument for functions returning a single scalar real value, i.e., set this value to the return value of the subroutine.

Table 2: SADScript Standard Argument List

To determine the actually number of arguments passed to your SADScript function you can inspect the difference between the global variable `isp`, which points to the top of the stack frame, and `isp1` which points to the stack frame base at the time of your function call. Letting `nArgs` denote the number of arguments passed to you function its value is computed according to

(1) `nArgs = isp – isp1`.

To extract the SADScript function arguments from the subroutine argument, and to return the subroutine results to the SADScript environment, we need to explore how the SAD stack frame is set up.

2 The SAD Stack Frame

SAD does much of its data passing using a stack. The stack is eight-bytes wide and is divided into "high" and "low" four-byte words. This layout is shown in Figure 1 for the case where SADScript function arguments are stored on the stack.

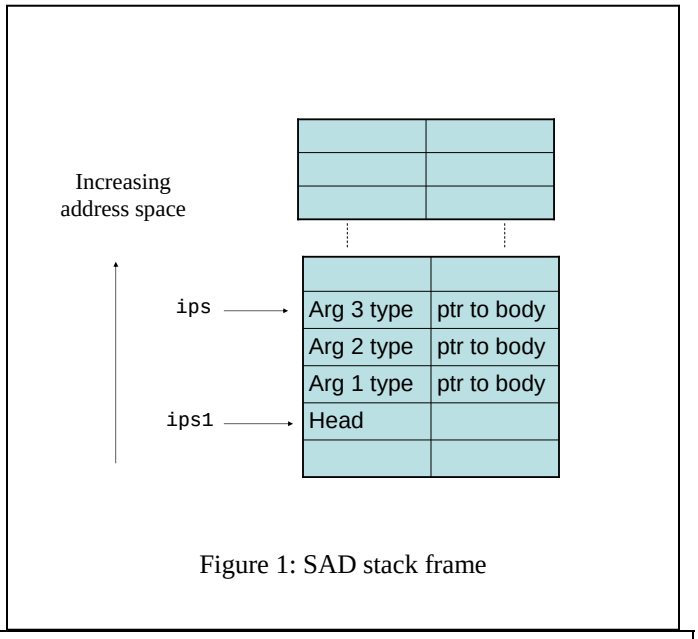

Figure 1: SAD stack frame

Global Variable	Type	Description
`Isp`	Integer*4	Pointer in the SAD stack frame ?
`ivstkoffset`		

Table 3: useful global variables in the SADScript environment

3 Including Your Modifications in the SAD Build

3.1 Source Files

In order for the SAD make system to include any source files in the build you should put them in the `src` subdirectory of the SAD repository (which is currently `oldsad`). Any FORTRAN source files placed here will automatically be compiled into object code. However, to include the object files in the build, you must make modifications to the object Makefile `oldsad/mk/sad.obj.mk`.

There are some environment variables you may wish to add to your .cshrc file if you are using the SAD repository main branch.

```
alias sad1  /usr/users/myuserid/oldsad/obj/OSF1/sad1.exe -c
setenv SAD_PACKAGES ~muserod/oldsad/Packages
setenv KBFRAMEDIR ~myuserid/oldsad/KBFrame
```

3.2 Modifying the Makefile

The SAD make system is composed of a set of make files distributed throughout the repository. The driver for all these files is located in the top directory `oldsad` have the default name `Makefile`. To include your object files in the SAD environment build you must add their file names and dependencies to the proper makefile. The dependent make files are included in the directory `oldsad/mk`. Depending upon with which CVS branch you are using, the name of the make is different. For the main trunk, this file is called simply `oldsad/mk/Makefile`. For the *amorita* branch of the repository the dependent make file is called `oldsad/mk/sad.obj.mk`.

Excerpt 5 is an example of this modification process. You must create a dependency so that `make` knows to include your object files in the build. In the example we have created a dependency called `objmine`. As you can see `objmine` depends upon the object files contained in the macro `OBJMINE`. The definition of this macro is given in the last line of the excerpt, we see that it contains the name of our example object file `MyFunctSub.o`.

```
all_object: obj0 obj1 obj2 obj3 obj4 obj5 obj6 \
            objdec objauto objca objutil objsim \
            objosdep objglue objdynl objmine MAIN.o

objmine: $(OBJMINE)

OBJ_LIBSAD=$(OBJ0) $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) \
    $(OBJ5) $(OBJ6) $(OBJAUTO) $(OBJF) $(OBJDEC) \
    $(OBJUTIL) $(OBJSIM) $(OBJCA) $(OBJRC) $(OBJTK) \
    $(OBJOSDEP) $(OBJENDIAN) $(OBJDL) $(OBJDYNL) \
    $(OBJMINE)

OBJMINE = MyFunctSub.o
```

Excerpt 5: makefile modifications