Ultra-Fast Macros: User-Written Functions Based on ADAMS/SDK

Trent Meehan
Mechanical Dynamics, Inc.
2301 Commonwealth Blvd
Ann Arbor, MI 48105
tmeeh@adams.com

Abstract

Customization of ADAMS work involves development of software in the three areas of ADAMS/View pre-processor, ADAMS/Solver, and ADAMS/View post-processor. The ADAMS/View macro language is the obvious choice for automation of many pre and post-processor tasks. However, for very large modeling processes there is a more efficient and higher-performance option - to develop ADAMS/View functions written in C and using the ADAMS/SDK (Software Developers Kit).

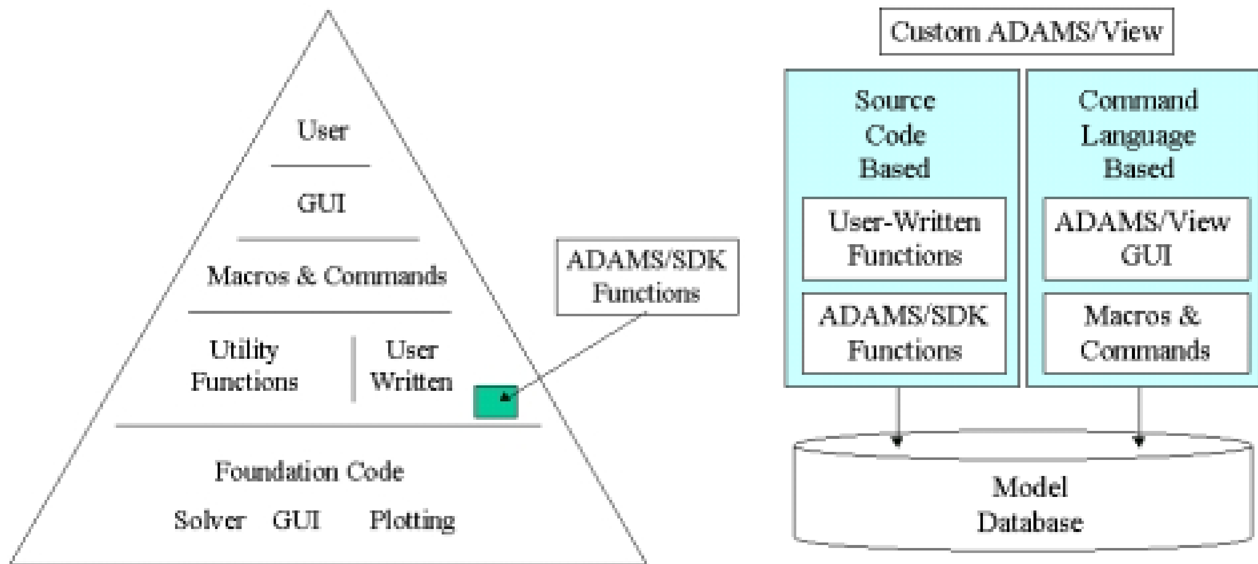
This paper provides a specific application showing how ADAMS/SDK is an enabling technology. A small and straight-forward example of source code will be provided to illustrate how this approach is ready for general use. This paper, combined with references and resources such as documentation of the ADAMS/SDK toolkit, will give readers confidence to consider this approach for their own applications.

Introduction

The ADAMS/View macro language is the proper choice for everyday automation of modeling tasks in ADAMS/View. But in extreme cases the customization of ADAMS/View resembles a software development effort, and the efficiency and speed of a source-code approach is needed. Certainly, work done in macros can be done in ADAMS/View user-written functions and compiled directly into the ADAMS/View executable.

But in order to replace a macro, a source code module must also be able to creation, access, modify, and delete modeling objects using a function call. Examples of modeling objects are, models, parts, markers, and their attributes; forces, joints, motions, and their attributes; graphics, gravity, and units; analysis sets, result sets, and result set components. Previously, this was a severe limitation to using ADAMS/View user-written functions alone. But now there is a way: ADAMS/View functions written in C call the functions of ADAMS/SDK to communicate directly with the ADAMS model database.

There are three steps to create ADAMS/View functions written in C using the functions of ADAMS/SDK. The three steps are 1) write, compile, and link a user-written function with ADAMS/View to create a ADAMS/View macro function; 2) write a function using ADAMS/SDK; and 3) join the ADAMS/SDK-based function with your user-written ADAMS/View macro function. Section 1 provides an example to demonstrate a user-written function in ADAMS/View. Section 2 discusses the relationship of ADAMS/SDK and ADAMS/View in ADAMS v10.1. Section 3 provides an example to demonstrate a user-written function with ADAMS/SDK function calls in ADAMS/View.



Sect ion 1 - Example User-Written ADAMS/View Macro Function

ADAMS/View documentation manuals "Running ADAMS on Windows" and " Running and Configuring ADAMS on UNIX" describe how to create a user-written ADAMS/View macro function. This technique allows the developer to perform difficult and lengthy numerical calculations and process inputs and outputs with the full power of a compiled language. This section shows how to create a user-written ADAMS/View function in a custom ADAMS/View executable, and use the function.

The example assumes Windows/NT and that ADAMS v10.1 and the appropriate F77 and C compilers are available.

Step 1. Write a user-written function

The following source file "vc_init_usr.c" is taken from the the file *install_dir*\aview\usersubs\vc_init_usr.c and edited to reduce clutter. It contains the user-written ADAMS/View function "add" and another function " vc_function_add ". The purpose of "vc_function_add" is to "register" the function "add" and make it available for use.

```
#include "mdi_c.h"

double add( double x, double y ) {
      return( x + y );
}

void vc_initialize_user( ) {
  /*              Function Function                              */
  /*              Name     Name          Argument  Return        */
  /*              Macro    Source        Types     Count  Units */
  vc_function_add("ADD",  (FUNCTION)add, fn_R_RR,  2,     0     );
}
```

Step 2. Compile and link with ADAMS/View

The following commands are issued to compile and link vc_init_usr.c into the dynamically linked library (DLL) "aview_user.dll".

```
cl /c /G6 /Ox /MD /I "C:\Program Files\ADAMS 10.1\aview\usersubs" vc_init_usr.c
adams101 aview cr-u n vc_init_usr.obj -n aview_user.dll -n
```

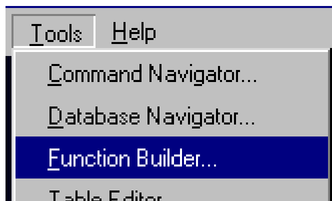
Step 3. Run the custom ADAMS/View executable

The following commands run the custom ADAMS/View DLL:

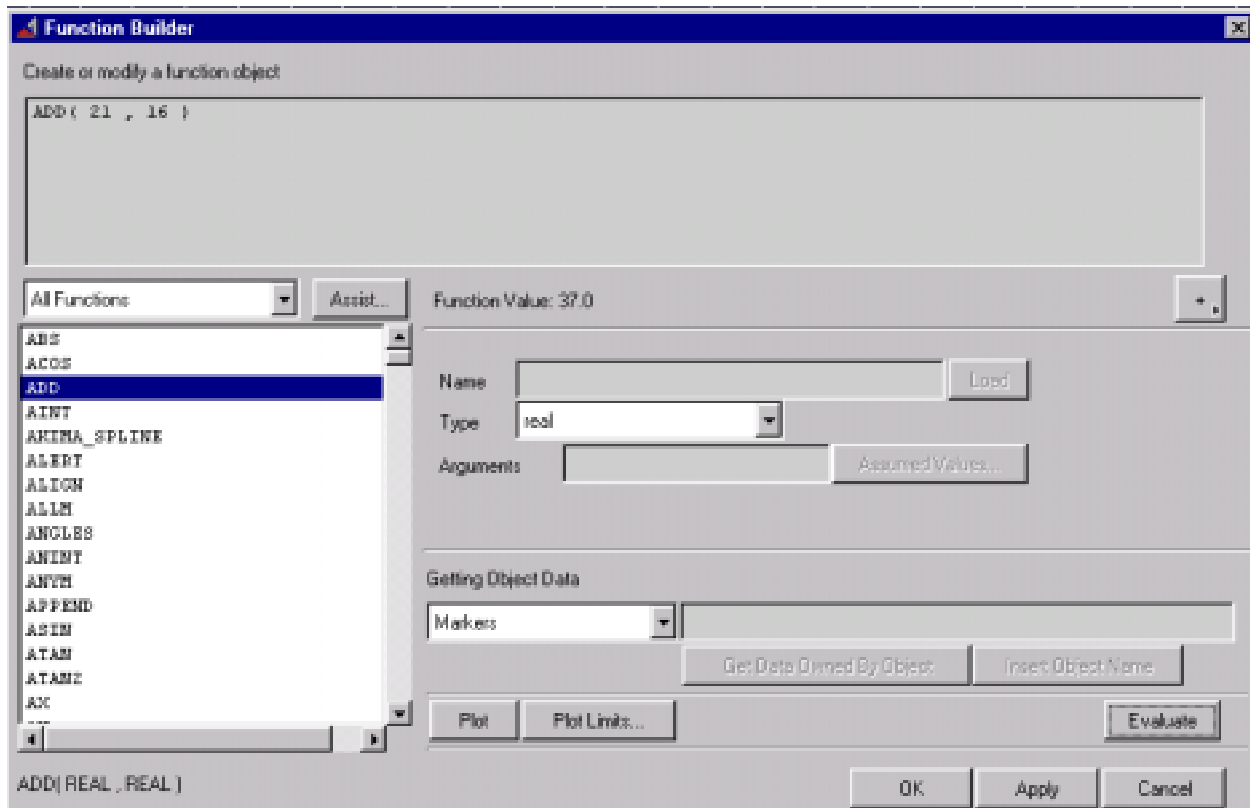
```
adams101 aview ru-u i aview_user.dll
```

Step 4. Use the user-written function in ADAMS/View

In the custom ADAMS/View session, the user-written function is available for use. For example, the new function is available in the function builder. The function add is put to work in the function builder to evaluate 21 + 16.



In the function builder panel: Choose "All Functions" -> Double click on "ADD" -> Enter 21 and 16 as the function parameters -> Click on "Evaluate".



Section 2 - Relationship of ADAMS/SDK and ADAMS/View

ADAMS/SDK enables creation, access, modification, and deletion of modeling objects directly from a source code level. ADAMS/SDK was developed for CAD/CAM/CAE partners who embed ADAMS capabilities, such as SDRC (I-DEAS), UG Solutions (Unigraphics), Engineering Animation (Vis-MockUp), Tecnomatix (Dynamo). Mechanical Dynamics also uses ADAMS/SDK for our embedded products with PTC (Pro/Engineer) and Dassault (CATIA). This is how ADAMS modeling objects can be generated and manipulated directly from within CAD environments.

The ADAMS manual "ADAMS/SDK DEVELOPERS' MANUAL" describes each function available through ADAMS/SDK in detail. ADAMS/SDK is intended as a way to embed the capabilities of ADAMS into CAD/CAM/CAE products. Therefore the audience for this manual is programmers who embed ADAMS into their own products using ADAMS/SDK. The ADAMS/SDK manual is a useful reference to understand the full scope of ADAMS/SDK. And there are several nice examples to show how these capabilities can be used.

Objects

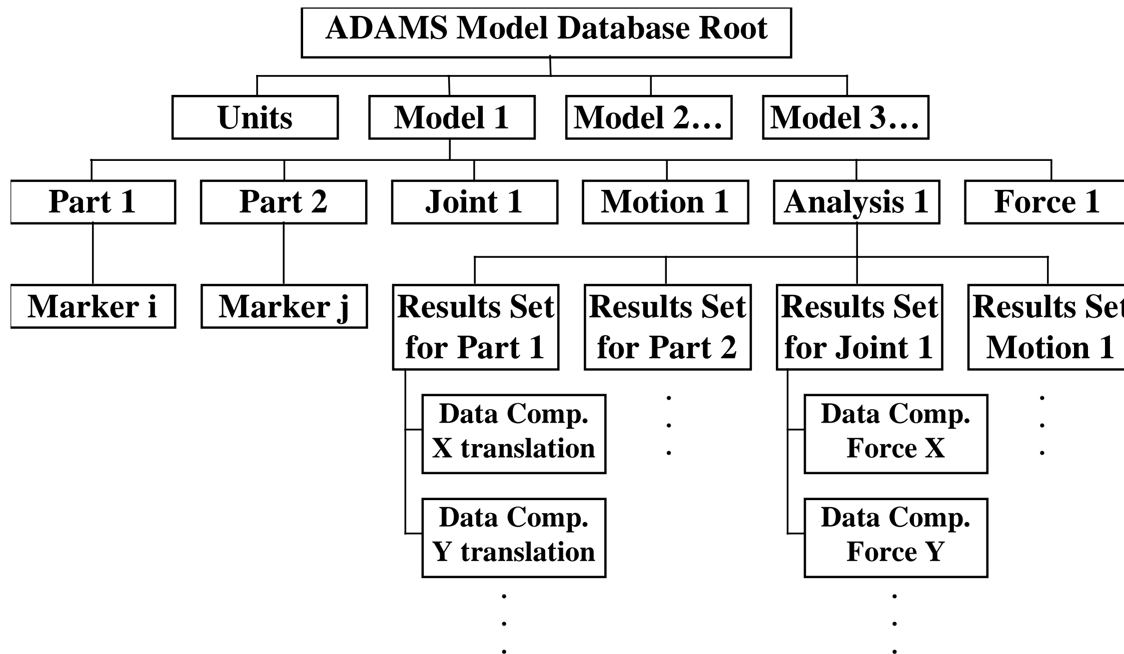
Objects in ADAMS/SDK correspond generally to the statements in the ADAMS dataset language, which have ADAMS ID numbers.

Attributes

An object in the database is described by the values of all of its attributes, including its object type, parent, and name, which are defined when it is created. The number and types of attributes that an object has depends on the type of object. The attributes for an object can be integer, real, Boolean, string, or object valued.

Hierarchy of ADAMS Objects

The ADAMS Model Database is a heirarchical database. The figure below illustrates the relationship between model objects and results objects. The figure depicts a mechanism with two parts, one joint, one motion input, and one force. One simulation has been run and the results saved in Analysis 1. The X and Y translations (as well as other motion outputs) of each part are stored in results set data components that belong to a results set that pertains to the part. There is one X translation stored for each part for each output step of the simulation.



Notice that the ADAMS Model Database Hierarchy hierarchy from the ADAMS/SDK documentation is **the same** as that in the ADAMS/View documentation. ADAMS/SDK operates on the same ADAMS Model Database which ADAMS/View uses. In fact, ADAMS/SDK functions are also available within ADAMS/View. Any ADAMS modeling element created through a ADAMS/View function call to ADAMS/SDK is immediately available within ADAMS/View.

After becoming familiar with the documentation, the programmer will tend to use the following header files as reference instead:

install_dir\\adamssdk\\include\\ac.h	- declarations of the functions
install_dir\\adamssdk\\include\\ac_objs.h	- enumerates the available objects
install_dir\\adamssdk\\include\\ac_atts.h	- enumerates the attributes that each object has

The objects and attributes in ac_objs.h and ac_atts.h are **the same** as those documented in the ADAMS/Solver manual. Therefore, the programmer will also use the ADAMS/Solver manual as a valuable reference.

Section 3 - Example User-Written ADAMS/View Macro Function Based on ADAMS/SDK

In ADAMS v10.1, the ADAMS/SDK library is provided together with ADAMS/View on the ADAMS delivery CD. Combining user-written ADAMS/View macro functions, ADAMS/SDK, and ADAMS/View macro language provides a powerful and efficient framework for design and maintenance of pre and post-processor software in ADAMS.

This section shows how to create an ADAMS model from within a user-written ADAMS/View function. The example is a pendulum model. This example goes beyond the ADAMS/SDK documentation by creating ADAMS graphics objects. The ADAMS/SDK documentation does not attempt to document creation of ADAMS graphics objects.

Remember that ADAMS/SDK was developed for CAD/CAM/CAE partners who embed ADAMS capabilities. Those packages have their own graphical elements. Not to worry. The ADAMS graphics objects and their attributes can be found in the ac_objs.h and ac_atts.h header files, and also in the ADAMS/Solver manual. In general, the graphical modeling objects which are available in ADAMS/SDK v10.1 are those supported in the ADAMS/Solver input (.adm) file. Box, cylinder, frustum, outline, and force graphics are in this category.

This example assumes Windows/NT and that ADAMS v10.1 and the appropriate F77 and C compilers are available. Appendix A provides a listing of the source files for this example, and lists the steps to run the example for yourself.

Step 1. Write a user-written function using ADAMS/SDK functions

Appendix A provides the source files for the ADAMS/View function that builds a pendulum model using calls to ADAMS/SDK functions.

`buildPendulum.c`	- user-written ADAMS/View function to create a pendulum model
`utils.c`	- functions to create ADAMS objects (ADAMS/SDK object creation functions are grouped together with the functions that set that objects attributes)
`utils.h`	- declaration headers for functions in utils.c
`vc_init_usr.c`	- "registers" the function " buildPendulum " and makes it available for use in ADAMS/View

Step 2. Compile and link function using ADAMS/SDK with ADAMS/View

The following commands are issued to compile and link vc_init_usr.c. The result is a dynamically linked library (DLL) "aview_user.dll".

```
cl /c /G6 /Ox /MD /I "C:\Program Files\ADAMS 10.1\aview\usersubs" /I
      "C:\Program Files\ADAMS 10.1\adamssdk\include" /I "." buildPendulum.c
      utils.c vc_init_usr.c

adams101 aview cr-u n buildPendulum.obj utils.obj vc_init_usr.obj -n
      aview_user.dll -n
```

Step 3. Run the custom ADAMS/View executable

The following commands were issued to run the ADAMS/View User DLL:

```
adams101 aview ru-u i aview_user.dll
```

Step 4. Use the user-written function in ADAMS/View

In section 1, the function builder invokes a user-written ADAMS/View. Here, the necessary commands commands are in an aview.cmd file in the local directory. Commands in the aview.cmd file are executed automatically upon startup of the custom ADAMS/View DLL. One of the commands in aview.cmd invokes the user-written ADAMS/View function "buildPendulum".

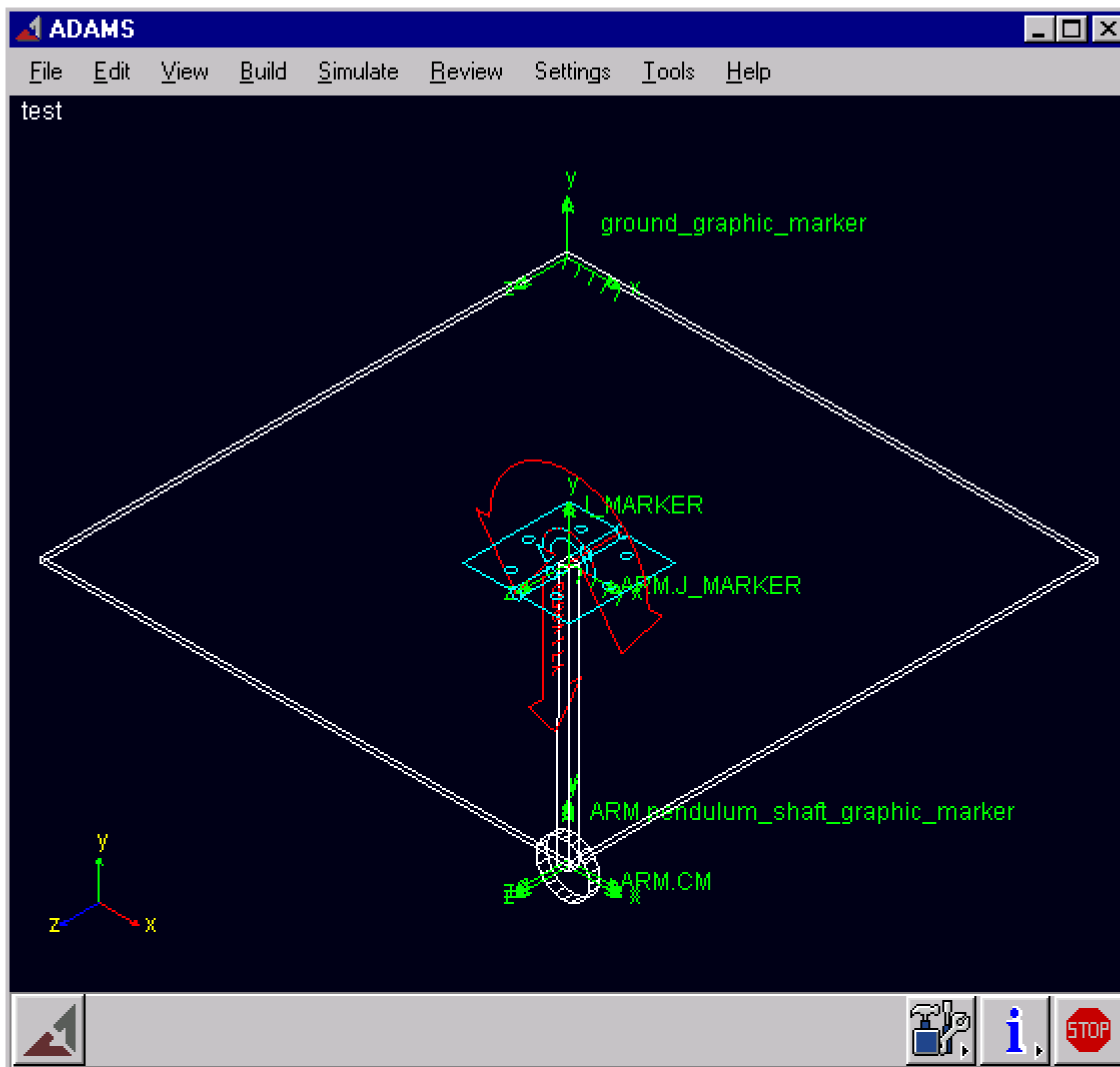
aview.cmd

```
def units length=meter mass=kg force=newton time=second angle=degrees
      frequency=hz
var set var=jint int=(eval(buildPendulum("test")))

!
!***** set visibility
!

entity attributes  &
 entity_name=.test &
 type_filter=model &
 visibility=no_opinion &
 name_visibility=no_opinion &
 transparency=0 &
 entity_scope=all_color

view zoom auto=on
```



Some comments relating to these example commands:

- Notice that the units are set from ADAMS/View. Then in buildPendulum.c, the units are set in ADAMS/SDK to be the same as the ADAMS/View settings. ADAMS/SDK expects the calling application to manage units (to set units and to provide model building data in those units).
- The user-written ADAMS/View command must pass the proper parameter type and accept the proper return value type. For this example, the input parameter is of type string, and the return value is of type integer.
- ADAMS modeling objects created by ADAMS/SDK functions do not have visibility attribute set. ADAMS/SDK expects the calling application to manage graphics. Therefore, the visibility of objects must be explicitly set by ADAMS/View after the objects have been created by ADAMS/SDK.

Benefits

The examples given in the paper are an introduction to the use of ADAMS/SDK within ADAMS/View user-written functions. The full power of this technique is beyond the scope of this paper. This technique may allow ADAMS to meet requirements previously dismissed as extreme. How this capability is applied is up the the imagination of the developer and the requirements of the organization.

The techniques described here have been applied with great success in a growing number of cases. An example application area is chain and belt model preprocessors. Detailed calculations are necessary to wrap the chain or belt. And these models have huge numbers of ADAMS modeling objects. Using source code to perform the numerical calculations and ADAMS/SDK to create the ADAMS objects has been shown to increase model creation speed of large models 100 fold.

Other custom preprocessors have been created to read parameter data from external files, perform detailed initial conditions calculations, and create huge numbers of ADAMS modeling entities. Postprocessors have been created to treat the output of ADAMS like channels of test data, perform data reduction, and create new result sets and populate them with the processed data.

Finally, it is a huge benefit to use the C language. Debugging tools are widely available for the C language. Development times decrease because C is well documented and understood. An organizations proprietary numerical methods can be incorporated into the pre and post processing. And more people are trained in standard compiled languages such as C.

Conclusion

The ADAMS/View macro language is the proper choice for everyday automation of modeling tasks in ADAMS/View. But in extreme cases, customization of ADAMS resembles a software development effort, and the ADAMS macro language may not be sufficient. In such cases, the ability to customize using a compiled language is needed to extend the capability of macros.

This paper describes how to combine user-written ADAMS/View macro functions written in C, ADAMS/SDK function calls, and ADAMS/View macro language. This technique provides the full power of customization using a compiled language together with flexibility of ADAMS/View macros. Appendix A provides a specific application with source code and instructions to help the reader get started.

References

1) Using ADAMS/Solver manual, Part number: 101SOLVUG-01, Statements and Function Expressions
2) Using the ADAMS/View Function Builder, Part number: 101VIEWFB-01, Expression Language Reference
3) Running ADAMS on Windows, Part number: 101NTRC-01, Creating Custom ADAMS/View Executable
4) "ADAMS/SDK DEVELOPERS' MANUAL", Part Number 10SIMDK-01
5) Header files on the ADAMS v10.1 CD
   a) *install_dir*\adamssdk\include\ac.h - declarations of the functions
   b) *install_dir*\adamssdk\include\ac_objs.h - enumerates the available objects
   c) \install_dir\adamssdk\include\ac_atts.h - enumerates the attributes that each object has

Appendix A: Source Files for the Example in Section 3

This section shows how to create an ADAMS model from within a user-written ADAMS/View function. The example is a pendulum model. The example goes beyond the ADAMS/SDK documentation by creating graphics. The graphic objects and their attributes can be found in the ac_objs.h and ac_atts.h header files, as described in the previous section, and also in the ADAMS/Solver manual. In general, the graphical modeling objects which are available in v10.1 are only those supported in the ADAMS/Solver input file. Box, cylinder, frustum, and outline graphics are in this category.

These steps will create and demonstrate this example application:
1. place the following source files in a single directory
2. edit the compile, link, and execute commands, if needed, to suit the particular installation of ADAMS on your computer platform
3. double click on cl_aview_v101.bat to compile the C source files and link with ADAMS/View
4. double click on run_aview_v101.bat to run the custom ADAMS/View DLL and execute the example using ADAMS/View commands from aview.cmd

BuildPendulum.c

```
#include <ac.h>
#include <mdi_c.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "pend.h"

int buildPendulum(char* model_name) {

      REAL             loc[3];
      REAL             vec[6];
      ADAMS_OBJECT     ao_model_id;
      ADAMS_OBJECT     ao_ground_id;
      ADAMS_OBJECT     ao_part_id;
      ADAMS_OBJECT     ao_force_id;
      ADAMS_OBJECT     ao_i_mark_id;
      ADAMS_OBJECT     ao_j_mark_id;
      ADAMS_OBJECT     ao_mark_id;
      ADAMS_OBJECT       ao_graph_id;
```

```
Char              label[200],fun[200];
Int               id,seg;
REAL              rad,len;

/* get units as set in ADAMS/View, set to the SDK */
ac_update_units();

/* first create an empty model, then add a ground part to it */
ao_model_id  = ac_create(ao_model, ADAMS_ROOT, model_name);
ao_ground_id = ac_create(ao_part, ao_model_id, "ground");
ac_set_object(ao_model_id, aa_model_ground, ao_ground_id);

/* define gravity as negative y direction in units of meters/sec**2) */
vec[0] = 0.0; vec[1] = -9.98; vec[2] = 0.0;
ao_force_id = ac_create(ao_accgrav, ao_model_id, "");
ac_set_reals (ao_force_id, aa_accgrav_vector, 3, vec);

/* add a marker to ground as a point to attach the joint */
ao_i_mark_id = ac_create(ao_marker, ao_ground_id, "I_MARKER");

/* add a marker to ground to attach ground graphic */
sprintf(label,"ground_graphic_marker");
id                = 0;
vec[0] = -5.0; vec[1] = 0.0; vec[2] = -5.0;
ao_mark_id = mk_mark(ao_model_id, label, id, ao_ground_id, 3, vec);

/* add a box graphic to represent ground */
sprintf(label,"ground_graphic");
vec[0] = 10.0; vec[1] = 0.1; vec[2] = 10.0;
ao_graph_id = mk_grap_box(ao_model_id, label, id, ao_ground_id,
      ao_mark_id, 3, vec);

/* create the arm of the pendulum and define its mass properties */
sprintf(label,"ARM");
id = 0;
vec[0] = 10.0; vec[1] = 5.0; vec[2] = 1.0; vec[3] = 5.0;
ao_part_id = mk_part(ao_model_id,label,id,3,vec,0,loc);

/* define the pendulum's center of mass marker */
sprintf(label,"CM");
id = 0;
loc[0] = 0.0; loc[1] = -5.0; loc[2] = 0.0;
ao_mark_id = mk_mark(ao_model_id, label, id, ao_part_id, 3, loc);

/* add a marker to pendulum to attach pendulum shaft graphic */
sprintf(label,"pendulum_shaft_graphic_marker");
id = 0;
vec[0] = -0.1; vec[1] = -5.0; vec[2] = -0.1;
ao_mark_id = mk_mark(ao_model_id, label, id, ao_part_id, 3, vec);

/* add a box graphic to represent pendulum */
sprintf(label,"pendulum_shaft_graphic");
vec[0] = 0.2; vec[1] = 5.0; vec[2] = 0.2;
ao_graph_id = mk_grap_box(ao_model_id, label, id, ao_part_id,
      ao_mark_id, 3, vec);

/* add a marker to pendulum to attach pendulum weight graphic */
sprintf(label,"pendulum_weight_graphic_marker");
id = 0;
vec[0] = 0.0; vec[1] = -5.0; vec[2] = -0.1;
ao_mark_id = mk_mark(ao_model_id, label, id, ao_part_id, 3, vec);
```

```
	/* add a box graphic to represent pendulum */
	sprintf(label,"pendulum_weight_graphic");
	id		= 0;
	rad		= 0.5;
	len		= 0.2;
	seg		= 20;
	ao_graph_id = mk_grap_cyl(ao_model_id, label, id, ao_part_id,
		ao_mark_id, rad, len, seg);

	/* add marker to pendulum as a point to attach the joint */
	sprintf(label,"J_MARKER");
	id = 0;
	ao_j_mark_id = mk_mark(ao_model_id, label, id, ao_part_id, 0, loc);

	/* create revolute joint between ground and pendulum */
	sprintf(label,"revjnt_grd_to_pend");
	id = 0;
	mk_revjnt(ao_model_id,label,id,ao_i_mark_id,ao_j_mark_id);

	/* create storque between ground and pendulum */
	sprintf(label,"storque_fun");
	sprintf(fun,"50*sin(2*time)");
	ao_force_id = mk_storque_fun(ao_model_id,label,id,0,ao_i_mark_id,
		ao_j_mark_id,fun);

	sprintf(label,"storque_fun_fgraphic");
	mk_grap_force( ao_model_id, label, id, ao_j_mark_id, ao_force_id);

	return 0;
}
```

utils.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ac.h>
#include <ac_adm.h>

/* make marker */
ADAMS_OBJECT mk_mark(
				ADAMS_OBJECT model,
				char* label,
				int id,
				ADAMS_OBJECT ao_part_id,
				int nvec,
				REAL vec[])
{
	ADAMS_OBJECT ao_mark_id;

	ao_mark_id = ac_create(ao_marker, ao_part_id, label);
	if( id ) ac_set_int(ao_mark_id, aa_adams_id, id);

	if( nvec==3 || nvec==6 ) ac_set_reals(ao_mark_id, aa_marker_location,
		3, &vec[0]);
	if( nvec==6 ) ac_set_reals(ao_mark_id, aa_marker_orientation,
		3, &vec[3]);
	if(!strcmp(label,"cm")||!strcmp(label,"CM"))
		ac_set_object (ao_part_id, aa_part_cm, ao_mark_id);

	return(ao_mark_id);
}
```

```
/* make part */
ADAMS_OBJECT mk_part(
                              ADAMS_OBJECT model,
                              char* label,
                              int id,
                              int nvec,
                              REAL vec[],
                              int nloc,
                              REAL loc[])
{
      ADAMS_OBJECT part;

      part = ac_create(ao_part, model, label);
      if( id ) ac_set_int(part, aa_adams_id, id);

      ac_set_real   (part, aa_part_mass, vec[0]);
      ac_set_real   (part, aa_part_ixx,  vec[1]);
      ac_set_real   (part, aa_part_iyy,  vec[2]);
      ac_set_real   (part, aa_part_izz,  vec[3]);
      if( nloc==3 || nloc==6 ) ac_set_reals(part, aa_part_location,
            3, &loc[0]);
      if( nloc==6 ) ac_set_reals(part, aa_part_orientation, 3, &loc[3]);
      return(part);
}

/* make joint */
ADAMS_OBJECT mk_revjnt(
                                ADAMS_OBJECT ao_model_id,
                                char* label,
                                int id,
                                ADAMS_OBJECT i_ao_mark_id,
                                ADAMS_OBJECT j_ao_mark_id)
{
      ADAMS_OBJECT ao_joint_id;

      ao_joint_id = ac_create(ao_revolute, ao_model_id, label);
      if( id ) ac_set_int(ao_joint_id, aa_adams_id, id);

      ac_set_object (ao_joint_id, aa_revolute_i, i_ao_mark_id);
      ac_set_object (ao_joint_id, aa_revolute_j, j_ao_mark_id);

      return(ao_joint_id);
}

/* make cylinder graphic */
ADAMS_OBJECT mk_grap_cyl(
                                    ADAMS_OBJECT model,
                                    char* label,
                                    int id,
                                    ADAMS_OBJECT ao_part_id,
                                    ADAMS_OBJECT ao_mark_id,
                                    REAL rad,
                                    REAL len,
                                    int seg )
{
      ADAMS_OBJECT ao_graph_id;

      ao_graph_id = ac_create(ao_graphic_cylinder, ao_part_id, label);
      if( id ) ac_set_int(ao_graph_id, aa_adams_id, id);

      ac_set_object  (ao_graph_id, aa_graphic_cylinder_cm, ao_mark_id);
```

```
	ac_set_real (ao_graph_id, aa_graphic_cylinder_radius, rad);
	ac_set_real (ao_graph_id, aa_graphic_cylinder_length, len);
	ac_set_real (ao_graph_id, aa_graphic_cylinder_angle, 360.0);

	ac_set_int (ao_graph_id, aa_graphic_cylinder_sides, seg);
	ac_set_int (ao_graph_id, aa_graphic_cylinder_segments, seg);

	return(ao_graph_id);
}

/* make box graphic */
ADAMS_OBJECT mk_grap_box(
				ADAMS_OBJECT model,
				char* label,
				int id,
				ADAMS_OBJECT ao_part_id,
				ADAMS_OBJECT ao_mark_id,
				int nvec,
				REAL vec[])
{
	ADAMS_OBJECT ao_graph_id;

	ao_graph_id = ac_create(ao_graphic_box, ao_part_id, label);
	if( id ) ac_set_int(ao_graph_id, aa_adams_id, id);

	ac_set_object  (ao_graph_id, aa_graphic_box_corner, ao_mark_id);
	ac_set_real (ao_graph_id, aa_graphic_box_x, vec[0]);
	ac_set_real (ao_graph_id, aa_graphic_box_y, vec[1]);
	ac_set_real (ao_graph_id, aa_graphic_box_z, vec[2]);

	return(ao_graph_id);
}


/* make single comp_torque */
ADAMS_OBJECT mk_storque_fun(
				ADAMS_OBJECT model,
				char* label,
				int id,
				int action_only,
				ADAMS_OBJECT i_ao_mark_id,
				ADAMS_OBJECT j_ao_mark_id,
				char* func)

{
	ADAMS_OBJECT ao_storque_id;

	ao_storque_id = ac_create(ao_storque, model, label);
	if( id ) ac_set_int(ao_storque_id, aa_adams_id, id);

		ac_set_object (ao_storque_id, aa_storque_i, i_ao_mark_id);
		ac_set_object (ao_storque_id, aa_storque_j, j_ao_mark_id);
		ac_set_bool(ao_storque_id, aa_storque_action_only, action_only);

		ac_set_string(ao_storque_id,aa_storque_func,func);

	return(ao_storque_id);

}
```

```
/* make vforce graphic */
ADAMS_OBJECT mk_grap_force(
                                       ADAMS_OBJECT ao_part_id,
                                       char* label,
                                       int id,
                                       int ao_marker_id,
                                       int ao_force_id)
{
      ADAMS_OBJECT ao_grap_id;

      ao_grap_id = ac_create(ao_graphic_force, ao_part_id, label);
      if( id ) ac_set_int(ao_grap_id, aa_adams_id, id);

      ac_set_object (ao_grap_id, aa_graphic_force_emarker, ao_marker_id);
      ac_set_object (ao_grap_id, aa_graphic_force_eobject, ao_force_id);

      return(ao_grap_id);
}
```

vc_init_usr.c

```
#include "ac.h"
#include "mdi_c.h"

int buildPendulum(char* aName);

void vc_initialize_user( ) {
   /*
    *             Function         Function
    *             Name             Name                        Arg     Return
    *             Macro            Source                      Types   Count
    */
   vc_function_add("buildPendulum", (FUNCTION)buildPendulum, fn_i_s, 1, 0);
}
```

cl_aview_v101.bat

```
cl /c /G6 /Ox /MD /I "C:\Program Files\ADAMS 10.1\aview\usersubs" /I
      "C:\Program Files\ADAMS 10.1\adamssdk\include" /I "." buildPendulum.c
      utils.c vc_init_usr.c
adams101 aview cr-u n buildPendulum.obj utils.obj vc_init_usr.obj -n
      aview_user.dll -n
```

run_aview_v101.bat

```
adams101 aview ru-u i aview_user.dll
```

aview.cmd

```
def units length=meter mass=kg force=newton time=second angle=degrees
frequency=hz
var set var=jint int=(eval(buildPendulum("test")))


!
!***** set visibility
!

entity attributes  &
 entity_name=.test &
```

```
 type_filter=model &
 visibility=no_opinion &
 name_visibility=no_opinion &
 transparency=0 &
 entity_scope=all_color

int grid undisplay
default force force_scale=1.0 torque_scale=1.0e3 display_text=yes
display_wireframe=yes
model attributes model=.test size_of_icons=1

view zoom auto=on
```