Solutions from HP EEsof

Building User-Defined Models

Product Note E4600-7

Overview

All Series IV Design Suites include the capability to incorporate user-developed models directly into a simulation by compiling the model with the simulation engines. This capability is provided by the HP E4638A Model Development Kit, which is an option to all Design Suites except OmniSys for which it is included.

This product note describes the use of the Model Development Kit with the Communications Design Suite (CDS) and Omnisys, which are Series IV system-level Design Suites. This powerful feature allows users greater flexibility in creating systems and applications requiring the integration of complex system algorithms and special purpose applications that are not available in CDS or OmniSys libraries.

User-defined models can be built for:

- customizing designs
- creating special purpose or proprietary models and libraries
- optimizing simulation speed
- porting coded algorithms to other platforms

This product note provides the methodology needed to incorporate user-defined models with examples of how to write, compile, and debug the code. Information on how to customize the models to the interface is also included.

Introduction

CDS and OmniSys require the use of an ANSI-C compiler and debugger when integrating models into the simulation engine. Most workstation computers are shipped with the standard Kernighan and Ritchie (KR) compilers, which are used for compiling and debugging the models, thereby allowing portability across platforms.

Source code and script files are provided in the *$EESOF_DIR/lib/omnisys/senior* directory. Make a project directory using CDS or OmniSys, or make your own local directory, and copy the following files.

- *omniindx.c* This code is used to link the modules contained in *omniproc.c* to interface with the main program (simulation engine).
- *omniproc.c* This code is used to register the user-defined model and the subroutines and external declarations to support the model.
- *senior.ael* This AEL code is used to register the model in a library or palette menu and can be used to define the model parameters with the interface.
- *omnisys_sr.make* This file is used by the UNIX *make* utility to control the compiler execution; this file describes the construction of the main program.
- *buildsr* This build script calls the *make* utility and makes the local links to access your new version of the simulator engine.

HEWLETT® PACKARD

Building a User-Defined Model

User-defined models are built by using a simple methodology. The methodology defines an 8-step process that is designed to allow users to first build the model and debug it, then integrate the model into the system model library. The process is:

- Define the algorithm
- Define memory requirements
- Code and debug the model
- Add the OmniSys database and messaging features
- Integrate the code into *omniproc.c*
- Make changes to *senior.ael*
- Compiling and debugging the simulator model
- Customize the model to the interface

Defining the Algorithm

The first step is to define the algorithm, the input and output, and timing used during the simulation of the model. The algorithm is an equation or set of parameters and constants that describe the operation of the model. The algorithm is evaluated over time; in this case time is a constraint of the operation of the model. The input and output must be known before starting the programming of the model because they define the model operation with other elements used in conjunction with the model and will represent the values passed to the model during the simulation. The parameters are values that are passed to the model before the start of the simulation and are used to change the operating characteristics of the model.

Figure 1 illustrates the process of defining the algorithm, input and output, timing and input parameters for a Walsh code generator. (Walsh code generators can be used for generating orthogonal codes in CDMA applications.) In this example we use 6-ary walsh code sequence meaning the model can generate 64 code sequences. Each code sequence contains 64 code states. The code state is similar to the phase angle in an analog signal. In this case, there are 64 phase states in each code sequence. The model is designed to allow the user to select 1 of 64 code sequences and 1 of 64 code phase states or a seed that is used to calculate a random phase state.

There are two methods for generating the algorithms: the first uses the product sum of a binary sequence; the second uses a simple table lookup method. The table lookup method is faster and is typically used where speed is a requirement.

Walsh code generator:

Equation:

$$WAL(n, t) = \prod_{r=0}^{p-1} (-1)^{(n_{(p-1-r)}) \bullet (i_r + i_{r+1})}$$

Initial Conditions:

p = 6

code sequence = N = 2^p; r,t=0 ... N – 1

or

Lookup table (WAL)

n = 31 -1,1,1-1,1,1 ...

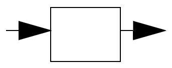
Limits: $\longleftarrow$ t

code sequence = 0 – 63

code phase = 0 – 63

I/O signal and parameter requirements for a Walsh code generator:

Clock(1,0)
Vin_i=clock
Vin_q=0



Clock(1,0)
Vout_i=code
Vout_q=0

Parameters:
code sequence (0 - 63, default = 32)
code phase (0 - 63, default = 0)
seed (0 - 32166, default = 0)

Figure 1. Defining the algorithm for a Walsh code generator

Defining Memory Requirements

The memory requirements phase is an important aspect of model design because all of the initial and run time states must be stored in memory. The memory requirements can be broken down into two parts:

- External States. The input and output signal states that represent the present or previous state.
- Internal States. The internal storage requirements of the model. These can be the input parameters to the model or constants and lookup table values.

For the model shown in Figure 1, the external requirements are: 1 of 2 clock states (1 or 0) and 1 of 64 code indices (0 to 63). The internal requirements are: the selected code state (1 of 64); and, the code phase (64 states). The total requirement is 66 (1 clock + 1 code + 64 phase states). Memory is allocated using the *malloc* utility and is usually stored as a pointer to a double precision value. Because there may be multiple instances of this model in your design, the OmniSys database utilities will be used to protect the memory stack used for each instance of the model; refer to the section “Adding the Database and Error Message Features.”

Coding and Debugging the Model

The model is written using the ANSI-C programming language. There are four basic tasks for building a user-defined model:

- Declaration of the model routine and initialization of the internal parameters and variables.
- Write the model.
- Write an executive routine to test the model.
- Debug the model to correct any errors and to improve the model design.

Writing the model begins with declaration of the model and any initialization phase. The declaration contains the type definition of the model routine (in this case the model must be defined as an integer or Boolean type to ensure compatibility with the simulator). The parameter values passed to the model also have a specific definition. Figure 2 shows an example declaration for a user-defined model including a set of model parameters.

```
int                 /* this type definition is later changed to boolean */
wal_gen(
  double vouti[][], /* I component of the output */
  double voutq[][], /* Q component of the output */
  double vini[][],  /* I component of the input */
  double vinq[][],  /* Q component of the input */
  double *time,     /* time instant of the simulation */
  double *tstep,    /* time increment of the simulation */
  char *pdat,       /* model parameter array */
  char *errstrg)    /* error string that is passed */
                    /* to the simulation if an error occurs */
```

NOTES:

1.

vouti/voutq	the I and Q component of the output voltage for a specific port number.
vini/vinq	the I and Q component of the input voltage for a specific port number.
time/tstep	the instant in time and the time step of the simulation.
pdat	passes the parameter data input from the user to the model.
errstrg	passes the error message back to the interface for processing.

2. The model is initially type cast as integer to allow the FALSE and TRUE signals to be passed back to the executive routine. Later we will type cast the model as boolean to make the model compatible with the interface. The input and output voltages will be changed from pointers to double precision pointers (double **).

Figure 2. User-defined model declaration

The signals are passed into the model by port reference and time instant. The input signals are passed to the model using *vini* and *vinq*, while the output signals are passed out to the simulator through *vouti* and *voutq*. The input and output signals are double precision arrays. The first index is a reference to the port number (the port number indexes are separate for both input and output, each begins with 0); the second index is a reference to the time instant of the sample (the time index is almost always 0 because the time index records the time in ascending order). Time and tstep represent the time instant of the simulation and the time increment or time step of the simulation. Parameters used to set up the behavior of the model are passed to the model using the double precision pdat array. At each instant in time the signals and parameters for the operation of the model are passed through the model interface; on completion the model must return a signal that indicates the outcome. If the calculations are successful, the model returns an integer value of one; if the model fails, a zero is returned to the simulator. The simulator uses the character string stored in *errstrg* to indicate an error by printing the message to the status panel.

The initialization occurs at time equals zero. Usually the first step in the initialization procedure is to check the values of the parameters passed into the model from the simulator interface. This is done to prevent erroneous parameter settings that can cause a segmentation fault or crash of the simulator. The values of each parameter are checked against the legal limits for that parameter. If the parameter is determined to be out of bounds, an error message is created and the model forces a return to the simulator. The signal for an error is to return with a FALSE or integer value of 0. The memory location of the error message is passed to the simulator using *errstrg*. Figure 3 shows an example of an error message. The maximum buffer length is 80 characters.

```
if(time == 0.0){ /* initialization */
  if(code_phase < 0.0 || code_phase > 32178.0){
  sprintf(errstrg,"%s","ERROR: Code_phase parameter not within legal limits");
  return FALSE;
}
```

Figure 3. Testing for bad input parameters

After the input parameter test the memory requirements are allocated and stored for the model. Memory allocation is the process of mapping the storage requirements for the parameters used during the operation of the model in memory. Allocation uses the UNIX utility *malloc* (memory allocation). Figure 4 shows an example of memory allocation.

```
code_sequence = pdat[0];      /* code sequence 0-63 (default=32)              */
code_phase = pdat[1];         /* code phase = 0-63 (default=0)                */
seed_value = pdat[2];         /* any 8 digit number greater than 0(default=0)   */
N = 64;                       /* N is the size of the walsh code sequence       */
if(time == 0){                /* at t = 0 initialize all variables             */
  numdata = N+2;              /* the 64 walsh codes                            */
  state_P = (double *)malloc((unsigned)sizeof(double)*numdata);
                              /* allocate space in memory for state_P and clk    */
```

Figure 4. Initializing the model at *time=0*

After the memory space has been allocated, the initial values can be stored. In Figure 5 the allocated space is initialized according to the input parameters (*code_sequence*, *code_phase* and *seed_value*). Other applications of allocated memory space would be the storage of previous states or values during the simulation of the model. (Later, the OmniSys database commands will be added that will protect against corruption from the multiple instance situation.)

```
if(code_phase == 0 && seed_value == 0){
/*code sequence selected ph=0 sd=0*/
  for(i=0; i<64; i++){
    state_P[i] = WAL[(int)code_sequence][i];
  }
}
```

Figure 5. Initializing allocated memory space

The initialization procedure usually ends with the return of the signal indicating the success or failure of the model as shown in Figure 6.

```
{
 Do initialization here...
return TRUE;
}
else{
  /* run time model goes here */
  return TRUE;
}
```

Figure 6. Using the definition *TRUE* to return the integer value 1 to signal success

After time equals zero the model begins looking like a state machine. Each input sample voltage will cause the output of the model to be calculated a certain way depending on the input signal levels and the input parameters associated with the model. Figure 7 illustrates the use of a clock signal as an input to the model. The clock signal is introduced on port zero (known as port one in the schematic representation of the symbol) and is used to step the model through a four-state machine. The model is triggered each time the clock rises from zero to one.

```
if(vini[0][0] == 0.0 && state_P[64] == 0.0){
  vouti[0][0] = 0.0;
  voutq[0][0] = 0.0;
  return TRUE;
} /* most active state - rising edge of clock signal */
if(vini[0][0] == 1.0 && state_P[64] == 0.0){
  state_P[64] = 1.0;/* update previous clock state */
/* calculate new value for vouti and voutq */
  vouti[0][0] = 1.0;
  voutq[0][0] = 0.0;
               return TRUE;
             }
```

Figure 7. Designing a state machine using the clock signal on port one (coded as port zero) of the schematic symbol

Creating the Executive Routine

Creating an executive routine to test the model code is important because the model must be evaluated prior to using it in the simulator. The executive routine is called “main” and is used to declare the variables used in the model and to initialize the parameters passed to the model from the executive routine. Figure 8 shows the construction of the main executive routine.

```
main()
{
  int i=0;
  double vini1[2][2], vinq1[2][2], vouti1[2][2], voutq1[2][2],
          time, tstep, pdat[3], timemax, signal[240], clk[240];
  char *errstrg;
  pdat[0] = 32.0; /* code sequence */
  pdat[1] = 0.0;   /* code phase */
  pdat[2] = 0.0;   /* seed value  */
  tstep = .001;
  time = 0.0;
  timemax = 100.0;
  while(time<=timemax){
    vini1[0][0] = clk[i++];
    vinq1[0][0] = 0.0;
    vouti1[0][0] = 0.0;
    voutq1[0][0] = 0.0;
    if(!wal_gen(vouti1,voutq1,vini1,vinq1,time,tstep,pdat,errstrg)){
      printf("%s\n",errstrg);
      exit(0);
    }
    printf("vout port 3 = %f\n",vouti1[0][0]);
    printf("vout port 4 = %f\n",vouti1[1][0]);
    time += tstep;
    if(i >= 240) i = 0;
  }
}
```

Figure 8. Using the code located in main executive routine to test the model (wal_gen)

Debugging the code is accomplished by compiling the executive and the model code using the -g option. This option maps each line of code so that the user can step through the code using a debugger at run time to look for the errors or bugs.

- For compiling the test program using SUN workstations:

  acc -g mytestprog.c -o mytestprog -lm

- For debugging the test program using SUN workstations:

  dbxtool mytestprog & <cr>

- For compiling the test program using HP workstations:

  cc -aA -g mytestprog.c -o mytestprog -lm

- For debugging the test program using HP workstations:

  xdb mytestprog <cr>

- Useful debugger commands on HP workstations:

  td toggle display (machine to ASCII)

  C continue execution

S step through the code

bp set initial breakpoints

b break on a line

db *line number*, delete break point on line

Adding Database and Error Message Features

OmniSys uses special commands to protect the memory allocated by your model. Each instance of the model is given an index so that the memory used by that instance is not overwritten by another instance. There are four commands that can be used to access the database at run time. (For more information, refer to the Series IV *User-Defined Elements* manual.)

- *omni_get_recind* Used to acquire the record index of the model instance. This command is inserted just before the initialization segment of the model (if (time == 0)). Once the record index is acquired, the model record can allocate memory, access memory or reallocate it.
- *omni_put_recdataP* Used during the initialization process to record memory allocated by the model instance.
- o*mni_get_recdataP* Returns a pointer to the data stored by *omni_put_recdataP* and the number of variables pointed to by it.
- *omni_realloc_dataP* Reallocates data. Used to change the size of the model data.

Figure 9 shows examples of OmniSys database commands.

```
numdata = N+1;              /* defines the number of memory locations needed */
recind = omni_get_recind(); /* gets record index number of element instance
                            in design */
state_P=(double *)omni_put_recdataP(recind, sizeof(double)*numdata, (void
*)state_P);                 /* put initial values of internal variables into
                            database for each instance of element */
state_P=(double *)omni_get_recdataP(recind, &numdata); /* get the number of
                            variables pointed to by state_P  */
```

Figure 9. OmniSys database commands examples

Error messaging can be handled using several different methods. The first method uses standard strings available to the user that are accessed by issuing an error code. The user-defined message and an error message indexed by the error code are assembled and displayed in the status panel. The second method (the most flexible because the user does not have to maintain external data structures) uses a message defined by the user and is displayed directly to the status panel. In the first version of the code, the model uses a string print or print function (*sprintf* or *printf*). These commands must be changed to one of the following:

- *get_omniproc_err* (first method). Uses *kwdindex* and optional user-defined error message. The *kwdindex* values are stored in the data structure *omniproc_err* and can be found in *omniproc.c*.
- *gparse_cpystr* (first and second methods). Used to send user-defined message to the status panel.

Figure 10 shows the usage of both commands.

Method 1
```
 gparse_cpystr("WAL_GEN",compname);
 gparse_cpystr("DSP_ELEMENT", kwdnam);
 gparse_cpystr("ERROR: I can’t remember!", errstrg);
 errind = 0; /* memory error */
 get_omniproc_err(compname, kwdnam, errind, errstrg); /* issues
   error message to status panel */
 return FALSE;
```
Method 2
```
 gparse_cpystr("ERROR: my foot is on fire!", errstrg); /* issues
   error message to status panel */
 return FALSE;
```

Figure 10. OmniSys error messaging utilities

Integrating the Code into *omniproc.c*

After the model code has been modified, it can now be integrated into the *omniproc.c* file. The 8-step process included in the *omniproc.c* file is defined as follows:

STEP 1 - edit lines in *omniproc.c* to define a new model. Define NUM_USER_COMPONENTS with the next available index number.

STEP 2 - insert the user-defined model functional interface.

STEP 3 - insert the user-defined model external declaration.

STEP 4 - modify the user component array by adding an entry for the new user-defined model.

STEP 5 - insert the user-defined model subroutines.

STEP 6 - insert the user-defined subroutine declarations.

STEP 7 - insert the user-defined model subroutines.

STEP 8 - insert the user-defined model code.

Steps 1 through 4 define the element; steps 5 through 7 define the subroutines used in the element model; Step 8 inserts the model code into the *omniproc.c* file. Figures 11 through 18 show the files for each of these steps.

Figure 18 shows the model code changes used to make the model compatible to the simulator interface. Note the changes to the model type cast (int to boolean), the changes to the input and output signal parameters (double array to double pointer) and the addition of the OmniSys database and message features. (The model example shown here is for a baseband DSP element. Consult the Series IV *User-Defined Elements* manual for information on the construction of electrical, functional, or optical models.)

```
#define NUM_USER_COMPONENTS 8 /* changed from 7 to 8 */
long NumUserComponents = NUM_USER_COMPONENTS;
#define PIPAD 0
#define CABLE 1
#define GAINPASS 2
#define GAINACT 3
#define GAINNL 4
#define DEMUB 5
#define LPFILTER 6
#define WALGEN 7 /* added component index value */
```

Figure 11. STEP 1, inserting numeric values

```
boolean f_pipad(struct fcomplex *, double *, double *, double *, char*);
boolean f_cable(struct fcomplex *, double *, double *, double *, char*);
boolean f_gainpass(struct fcomplex *, double *, double *, double *, char*);
boolean f_gainact(struct fcomplex *, double *, double *, double *, char*);
boolean f_gainnl(struct fcomplex *, double *, double *, double *, char*);
boolean f_demsb(double *, double *, double **, double *, long *, double **,
          long *, int, int, boolean, double *, char*);
boolean lpfilter(double **, double **, double **, double **, double, double,
          double *, char*);
boolean wal_gen(double **, double **, double **, double **, double, double,
          double *, char*); /* added this declaration for the wal_gen model */
```

Figure 12. STEP 2, inserting interface declarations

```
extern boolean f_pipad();
extern boolean f_cable();
extern boolean f_gainpass();
extern boolean f_gainact();
extern boolean f_gainnl();
extern boolean f_demsb();
extern boolean lpfilter();
extern boolean wal_gen(); /* added this external declarartion for the wal_gen model */
```

Figure 13. STEP 3, inserting external declarations

```
struct user_def_component UserComponent[NUM_USER_COMPONENTS] = {
{"WALl_GEN",
      "Walsh code generator",
      wal_gen,
      NULLPOINTER,
      ELEMENT_DSP,
      2,
      1,
      3,
      “code sequence”,
      "code_phase",
      "seed_value"}};
```

Figure 14. STEP 4, inserting entries into UserComponent array

```
void copy_complex(struct fcomplex *, int, struct fcomplex *);
int rand_val(int);
```

Figure 15. STEP 5, inserting subroutine declarations

```
extern boolean void copy_complex();
extern int rand_val(); /* added this external reference for the subroutine rand_val */
```

Figure 16. STEP 6, inserting subroutine external declarations

```
void copy_complex(struct fcomplex *in, int n, struct fcomplex *out){
  int i;
  for(i = 0; i < n; i++){ out.re[i] = in[i].re; out[i].im = in[i].im;}
}
int /* added this code */
rand_val(int seed)
{
  int stime, count=0;
  double val;
  long ltime;
  ltime = time(NULL);
  stime = (unsigned int)ltime/2;
  srand(stime+seed);
  for(;;){
    val = rand()/1000;
    if(count++ > 50)break;
    if(val <= 63)break;
  }
  return val;
}
```

Figure 17. STEP 7, inserting model subroutines

```
boolean
walsh_gen(
double  **vouti,  /* caution: be careful when implementing the order  */
double  **voutq,  /*          of the arguments for the element model  */
double  **vini,   /*          elements using fc maybe handled diff-   */
double  **vinq,   /*          erently!                                */
double  time,     /* Note: vini[port number][time index]              */
double  tstep,    /*       vout[port number][time index]              */
double  *pdat,
char *errstrg)
{
  double *state_P, code_sequence, code_phase, seed_value;
  extern int WAL[64][64];
  int r_val, j;
  long recind, i, numdata, N, count;
  code_sequence = pdat[0];      /* code sequence= 0-63 (default=32)           */
  code_phase = pdat[1];         /* code phase = 0-63 (default=0)               */
  seed_value = pdat[2];         /* any 8 digit number greater than 0(default=0)*/
  N = 64;                       /* N is the size of the walsh code sequence    */
  recind = omni_get_recind();   /* record number for the instance       */
                                /* of the element in the design         */
  if(time == 0){                /* at t = 0 initialize all variables    */
    numdata = N+2;              /* the 64 walsh codes                   */
    state_P = (double *)malloc((unsigned)sizeof(double)*numdata);
                                /* allocate space in memory for state_P and clk */
    if(code_sequence < 0 || code_phase < 0 || seed_value < 0){
      gparse_cpystr("Code sequence, phase or seed is less than zero",errstrg);
      return(FALSE);            /* error message to catch parameters < 0          */
    }
    if(code_sequence > 63 || code_phase > 63 || seed_value > 32000){
      gparse_cpystr("Code sequence > 63, or phase > 63 or seed > 32000",errstrg);
      return(FALSE);            /* error message to catch parameters > 63 or 32K  */
    }
    if(code_phase > 0 ){        /* code phase is set to a selected sequence    */
      count = 0; j = code_phase;
      for(i=0; i<64; i++){      /* load all 64 code states for selected seq */
        state_P[i] = (double)WAL[(int)code_sequence][j];
        j++;
        if(j == 64 && count ==0){j=0; count++;}
      }
    }
    if(code_phase == 0 && seed_value == 0){/*code sequence selected ph=0 sd=0 */
      for(i=0; i<64; i++){
        state_P[i] = (double)WAL[(int)code_sequence][i];
      }
    }
    if(code_phase == 0 && seed_value > 0){/* calc seed value output random ph */
      r_val = rand_val(seed_value);
      count = 0; j = r_val;
      for(i=0; i<64; i++){
        state_P[i] = (double)WAL[(int)code_sequence][j];
        j++;
        if(j == 64 && count ==0){j=0; count++;}
      }
    }
    i=64;
    state_P[i++] = 0;          /* entry for clock state state_P+64               */
    state_P[i] = 0;            /* entry for code state state_P+65                */
                               /* assign initial values to output of instance    */
    vouti[0][0] = *(state_P + (int)(*(state_P+65)));
    voutq[0][0] = 0.0;
```

Figure 18. STEP 8, inserting the user-defined model code (1 of 2)

```
      /* put initial values into database for this instance of the element        */
      /* caution: it is not good programming practice to get and put value        */
      /*          into the database at the same time instant! This may            */
      /*          corrupt the database values                                      */
      omni_put_recdataP(recind, sizeof(double)*numdata, (void *)state_P);
    } /* End of initialization tasks */
    else {
  /* at time> 0 get values from database for this instance of the element        */
  /* check clock state with vini[0][0], if state goes from zero to one           */
  /* change state_P, if clock state goes from one to zero save state only        */
      state_P = (double *)omni_get_recdataP(recind, &numdata);
      /* clock transition from 0 to 1                          */
      if(state_P[64] == 0.0 && vini[0][0] >= 1.0){
        if(state_P[65] == 64){
          state_P[65] = 0;
        }
        if(state_P[65] != 64.0){
          state_P[65] = state_P[65] + 1;
        }
        vouti[0][0] = state_P[(int)state_P[65]];
        voutq[0][0] = 0.0;
        state_P[64] =  1.0;
        return( TRUE);
      }
      /* clock transition from 1 to 0                          */
      if(state_P[64] == 1.0 && vini[0][0] < 1.0){
        vouti[0][0] = state_P[(int)state_P[65]];
        voutq[0][0] = 0.0;
        state_P[64] = 0.0;
        return( TRUE);
      }
      /* no transition in clock state 0 to 0                   */
      if(state_P[64] == 0.0 && vini[0][0] == 0.0){
        vouti[0][0] = state_P[(int)state_P[65]];
        voutq[0][0] = 0.0;
        return( TRUE);
      }
      /* no transition in clock state 1 to 1                   */
      if(state_P[64] == 1.0 && vini[0][0] >= 1.0){
        vouti[0][0] = state_P[(int)state_P[65]];
        voutq[0][0] = 0.0;
        return( TRUE);
      }
    }
  }
```

Figure 18. STEP 8, inserting the user-defined model code (2 of 2)

Making Changes to *senior.ael*

The *senior.ael* file defines the physical symbol and parameters used to represent the model in the system schematic. The symbol consists of a drawing and parameter set. Examples of the AEL code used to define the symbol are located in the *senior.ael* file; the Series IV *AEL Guide* contains definitions and examples of the parameter for each of the commands used in this example. There are three steps for defining the user-defined model symbol in the *senior.ael* file. First, the user must edit the *senior.ael* file and enter a *create_item* command for the model. Second, the menu and palette lists must be added. Third, a directory for the AEL file must be created and the *.simframe* file changed to note the location of the AEL file.

Adding the *create_item* line for the model is shown in Figure 19. Each of the arguments used in the *create_item* command have some significance, but may not have anything to do with creating a symbol for the user-defined model. (For more details regarding these commands refer to the Series IV *AEL Guide*.)

Command definitions

```
create_item(name, label, prefix, attrib, priority, iconName,
        dialogCode, dialogData, netlistFormat, netlistData,
        displayFormat, symbolName, artworkType, artworkData, parameterN)
create_parm(name, label, attrib, formSet, unitCode, defaultValue)
```

Example of *create_item* and *create_parm*

```
create_item("WAL_GEN", "Walsh Code Generator", "D", NULL, NULL, NULL,
        standard_dialog,  "Digital Model",  standard_netlist, "ELEMENT",
standard_symbol, "2PORT", no_artwork, NULL,
create_parm("code_sequence", "Walsh code 0-63", 0, "rvopt", 0, 31.0),
create_parm("code_phase", "Code phase 0-63", 0, "rvopt", 0, 0.0),
create_parm("seed_value", "Random generated phase 0-32176", 0, "rvopt", 0,
       0.0));
```

Figure 19. Adding the *create_item* line in *senior.ael*

The first argument is the *name* parameter, which represents a unique name given to describe the model. The second argument is the *label* parameter, which often describes the function of the model. The third argument is the *prefix* parameter, which is the letter associated with the tag given to the symbol when the element is placed into the schematic. For this model the *prefix* letter is D, which stands for DSP element. The next two arguments, the *attrib* and the *priority* parameters are not discussed here. The next argument is the *iconName* parameter, which can be used to specify the name of a bit map for the element in the palette menu (this argument is discussed later in the section “Customizing the model to the user interface”).

The next five arguments, *dialogCode*, *dialogData*, *netlistFormat*, *netlistData* and *displayFormat* are not discussed. The next argument is called *symbolName* and can be used to define custom artwork. In this case a default representing the number of ports used in the model has been used (2PORT). The next two arguments *artworkType* and *artworkData* are not discussed.

The last set of arguments are nested commands called *create_parm*, which are used to define the parameters that will be displayed with the schematic symbol. In this case there are three parameters, *code_sequence*, *code_phase* and *seed_value*. The *create_parm* command arguments are similar to the arguments used in the *create_item* command. The first argument is the *name* of the parameter, the second is a descriptive *lable*. The next argument is called the *attrib* and is not discussed here. The next three arguments control how the parameter value is read in to the simulator. The *formSet* argument is used to define the type of parameter (numeric or string). The *unitCode* argument defines the units associated with the parameter value. The *defaultValue* argument specifies what is displayed when the symbol is placed in the schematic.

The schematic and test bench library menu lists use the commands *set_design_type*, *library_group* and *palette_group* to create the menu and palette lists for the schematic and test bench interfaces. The *set_design_type* command instructs the AEL command interpreter to create the menu in the schematic or the test bench interface. The *palette_group* and *library_group* are used to distinguish the button palette (used to select the model from a series of buttons on the left hand side of the interface) from the library menu (accessed through the menu button at the top of the schematic page that creates the menu dialog box). The *set_design_type* is used to define the location of the menu, 31 is the location of the schematic and 41 is the location of the test bench. The palette and library group commands have the same arguments. The first argument is the name of the menu. The second argument is the label used in the main menu. The remaining arguments are the model names that allow the user to select and place the symbol onto the schematic. Figure 20 shows an example of the menu and palette definitions.

Command definitions
```
set_design_type (location);
palette_group (name, label, menu_item1,......menu_itemN);
library_group (name, label, menu_item1,......menu_itemN);
```

Examples of *set_design_type, palette_group* and *library_group* AEL commands
```
set_design_type (31); /* 31 for the schematic 41 for the testbench */
palette_group ( "senior" , "Senior elements", "PIPAD", "CABLE",
        "GAINPASS", "GAINACT", "GAINNL", "DEMUSB",
        "LIMITER", "iqGRAY", "TGAIN", "DGAIN","WAL_GEN" );
library_group ( "senior" , "Senior elements", "PIPAD", "CABLE",
        "GAINPASS", "GAINACT", "GAINNL", "DEMUSB",
        "LIMITER", "iqGRAY", "TGAIN", "DGAIN","WAL_GEN" );
set_design_type (41);
palette_group ( "senior" , "Senior elements", "PIPAD", "CABLE",
        "GAINPASS", "GAINACT", "GAINNL", "DEMUSB",
        "LIMITER", "iqGRAY", "TGAIN", "DGAIN","WAL_GEN");
library_group ( "senior" , "Senior elements", "PIPAD", "CABLE",
        "GAINPASS", "GAINACT", "GAINNL", "DEMUSB",
        "LIMITER", "iqGRAY", "TGAIN", "DGAIN",WAL_GEN" );
```

Figure 20. Creating the menu list and palettes

The third step in making changes to the AEL code are done to ensure the AEL code is read during start-up of the design environment. Edit the *.simframe* file in your project directory to enter the location of the *senior.ael* file. To make the file accessible to everyone, you can create an AEL directory anywhere. Create a project directory for OmniSys (*omnisys*) or CDS (*comms*) and move the *senior.ael* file to the appropriate location. Edit the *.simframe* file in your project directory and place the line *USER_AEL=yourpath/ senior.ael*. For the AEL code to be local to your project, place the *senior.ael* file in your network directory. Compiling the AEL code to check for errors can be accomplished by running the AEL compiler as shown in Figure 21. The AEL code will be read in when the design environment is started. Figure 22 shows an example of the use of the *USER_AEL* directive.

Definition
```
aelcomp source_file_name destination_file_name
```
Example
```
aelcomp senior.ael senior.atf
```

Figure 21. AEL compiler command line usage example

```
IS_PROJECT_DIR=TRUE
#when using $HOME/ael/comms or $HOME/ael/omnisys
USER_AEL=senior.ael
#when using the local project directory
USER_AEL=./senior.ael
```

Figure 22. *.simframe_comms* file showing the use of *USER_AEL*= directive

Compiling and Debugging the Simulator Model

It is necessary to debug the model if problems arise during the simulation. To compile a debug version of the code that can be run inside the debugger, first edit the make file called *omnisys_sr.make*. Edit the line that defines the compiler flags—the variable used is called CFLAGSP. The flag is usually set to -O, which is used to generate an optimized version of the code. Change this flag to a -g option. You can now run the *buildsr* script, which invokes the make utility and runs the compiler.

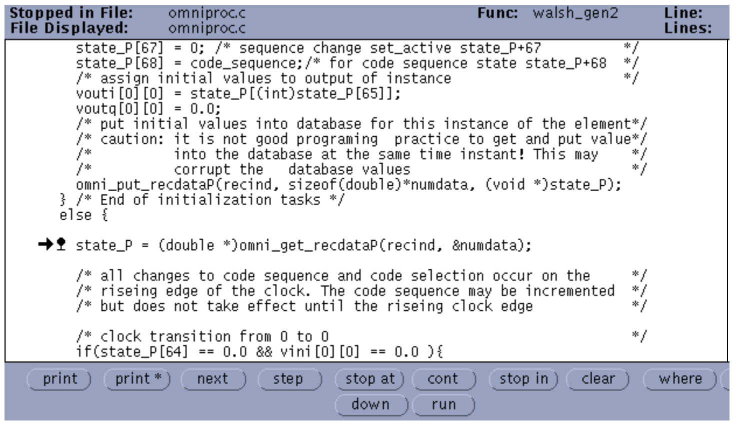
Compiling the model begins with running the *buildsr* script file. The *buildsr* compiles *omniindx.c* and *omniproc.c* and builds a new simulator by linking the model to the simulator. Debugging the simulator can be done by running the design environment and simulating a working test bench. This starts the simulator in the background. You must get the process ID of the simulator by running the process status command (ps) and looking for *omnisys.bin*. The process ID is used to attach the debugger to the simulator engine process. This is accomplished by starting the debugger with the engine name and the process ID on the command line

- for SUN workstations: **dbxtool omnisys.bin 4215 &<cr>**
- for HP workstations: **xdb omnisys.bin -P4215 <cr>**

The dynamics of the debuggers are quite different— consult the workstation documentation on the use of the debugging tools. Placing the model in the schematic and running the simulation again will allow the simulation to run the model so you can view the model execute in the run time environment as shown in Figure 23. At simulation time the simulation may stop initially, this is remedied by issuing the continuation command on the command line of the debugger you are using.

- for SUN workstations the command is **cont**
- for HP workstations the command is **C**

This signals the engine to continue with simulation.



```
Stopped in File:    omniproc.c                                Func:  walsh_gen2      Line:
File Displayed:     omniproc.c                                                       Lines:
        state_P[67] = 0; /* sequence change set_active state_P+67          */
        state_P[68] = code_sequence;/* for code sequence state state_P+68  */
        /* assign initial values to output of instance                     */
        vouti[0][0] = state_P[(int)state_P[65]];
        voutq[0][0] = 0.0;
        /* put initial values into database for this instance of the element*/
        /* caution: it is not good programing  practice to get and put value*/
        /*           into the database at the same time instant! This may    */
        /*           corrupt the   database values                           */
        omni_put_recdataP(recind, sizeof(double)*numdata, (void *)state_P);
      } /* End of initialization tasks */
      else {

    ➜●  state_P = (double *)omni_get_recdataP(recind, &numdata);

        /* all changes to code sequence and code selection occur on the     */
        /* riseing edge of the clock. The code sequence may be incremented  */
        /* but does not take effect until the riseing clock edge            */

        /* clock transition from 0 to 0                                     */
        if(state_P[64] == 0.0 && vini[0][0] == 0.0 ){
```

Figure 23. Running the debugger to evaluate the user-defined model

Customizing the New Model to the User Interface

The new user-defined model can be customized by providing a symbol that is more descriptive of the model's function and usage. Create the schematic symbol in the design environment by using the symbol view and symbol editing features available under the schematic editor in the design environment. The bitmaps for the button palettes can be generated using the bit map icon editors available in the workstation window environments:

- *vueicon* for HP workstations
- *iconedit* for SUN workstations

Start by editing a bit map using the icon editor under your workstation environment. The bit map examples that are compatible with the design environment can be used as a starting point in the editing process.The bit map examples are located under *$EESOF_DIR/lib/omnisys/bitmaps*. Store the edited bit maps under your own bit map directory (*$HOME/bitmaps*).

Figure 24 shows the symbol being generated for this example.

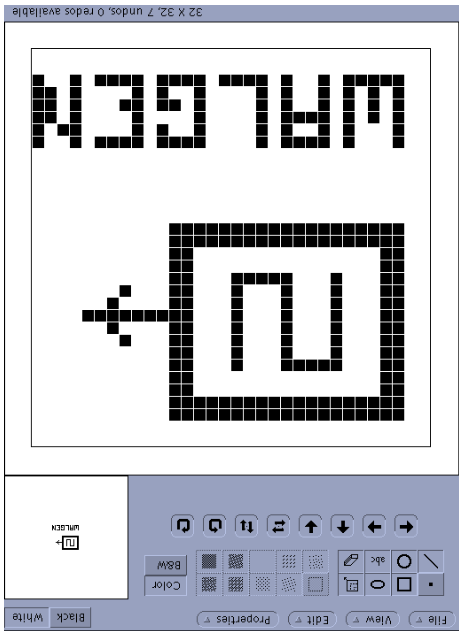


Figure 24. Defining a bit map for the button palette

To build the symbol using the schematic editor, start a new design in the design environment and switch to the symbol view. As shown in Figure 25, the symbol is constructed using the symbol editing features in the DRAW menu. Draw the symbol using the editor functions, add the text and symbol pins, and save the symbol. Add the file names of the bit map and schematic symbol in the line containing the *create_item* command in the *senior.ael* file, as shown in Figure 26. Edit your *senior.ael* file: change the *iconName* argument to the filename with your bit map; change the *symbolName* argument to the new symbol name; save the file.

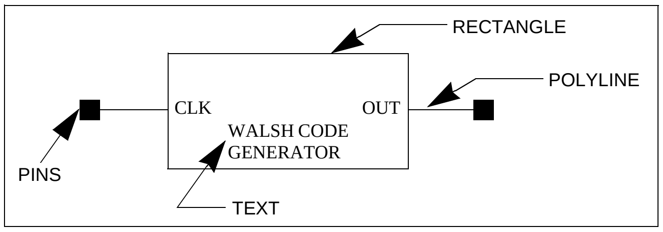
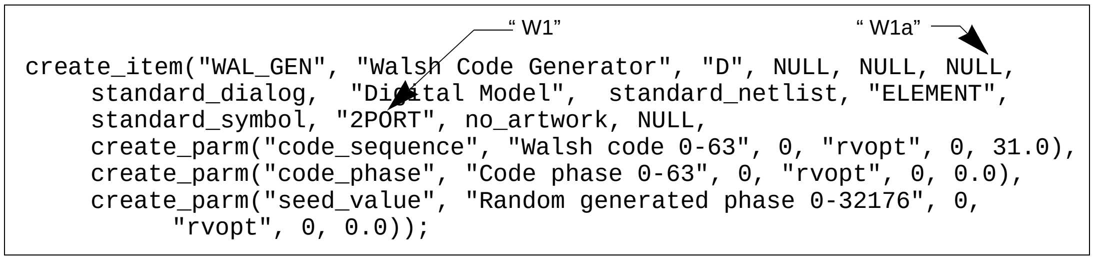


Figure 25. Building a schematic symbol



```
create_item("WAL_GEN", "Walsh Code Generator", "D", NULL, NULL, NULL,
     standard_dialog,  "Digital Model",  standard_netlist, "ELEMENT",
     standard_symbol, "2PORT", no_artwork, NULL,
     create_parm("code_sequence", "Walsh code 0-63", 0, "rvopt", 0, 31.0),
     create_parm("code_phase", "Code phase 0-63", 0, "rvopt", 0, 0.0),
     create_parm("seed_value", "Random generated phase 0-32176", 0,
           "rvopt", 0, 0.0));
```

Figure 26. Editing the senior.ael file to add the filenames of the bit maps (W1a.bmp) and symbols (W1.dsn) under *iconName* and *symbolName*. The names are added to the argument list without the filename extensions.

HEWLETT®
PACKARD

For more information, contact a regional HP office listed below. Or check your telephone directory for a local HP sales office.

United States
800-452-4844

Canada
905-206-4725

Europe (Amsterdam)
European Marketing Center
P.O. Box 999
1180 AZ Amstelveen
The Netherlands

Japan
81-426-48-0237

Latin America
(Miami, Florida)
305-267-4245

Australia/New Zealand
13-1347 ext. 2902

Asia Pacific (Hong Kong)
8522-599-7070

Data subject to change

© 1996 Hewlett-Packard Company
Printed in USA PN E4600-7
1/96 , 2.5C   3/96 2.5C

5964-9003E