Infor CloudSuite Industrial Scheduler Customization Guide

10/30/15

Important Notices

The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.

By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement, the terms of which separate agreement shall govern your use of this material and all supplemental related materials ("Purpose").

In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above. Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Without limitation, U.S. export control laws and other applicable export and import laws govern your use of this material and you will neither export or re-export, directly or indirectly, this material nor any related materials or supplemental information in violation of such laws, or use such materials for any purpose prohibited by such laws.

Trademark Acknowledgements

Publication Information

Release: Infor CloudSuite Industrial 9.00.x

Publication date: October 30, 2015

Contents

About This Guide

This manual provides instructions for customizing the Scheduler, i.e. creating user-defined rules. It begins with a discussion of how to write User Defined Rules with logic not included in the standard system and then presents several sections in reference format which describe the system modeling constructs used to customize the Scheduler.

There are 8 different rule types that can be user defined. In the drop down list on the form you will see something like “User Defined Allocation Rule 12, etc. That will indicate the user defined rule numbers that you can create. Following is a table which contains each type of rule, where it is found on the forms, what numbers are allowed, and how it fits into the Scheduler’s logic flow. Following that table is a figure which shows how a job progresses through the system from the Scheduler’s rule point of view.

Intended audience

This information is intended for advanced APS users.

Contacting Infor

If you have questions about Infor products, go to the Infor Xtreme Support portal at http://www.infor.com/inforxtreme.

If we update this document after the product release, we will post the new version on this Web site. We recommend that you check this Web site periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

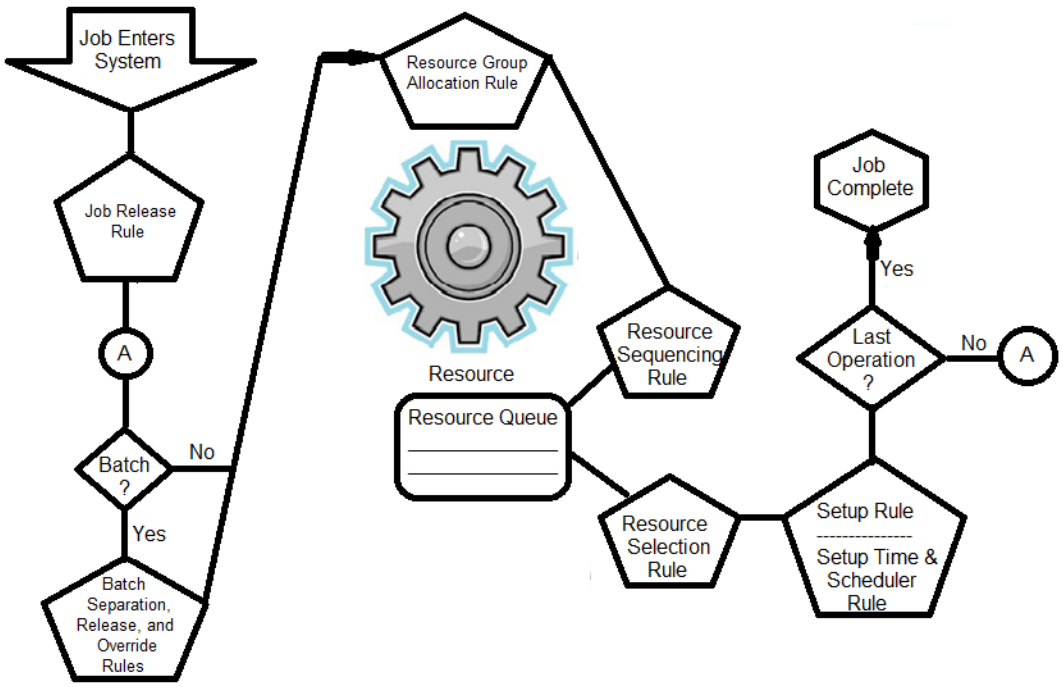
Chapter 1: Overview

1

We will begin by discussing the Scheduler's flow of logic and where rules come into play. There are 8 different rule types that can be user defined. In the drop down list on the form you will see something like "User Defined Allocation Rule 12," etc. That will indicate the user defined rule numbers that you can create. Following is a table which contains each type of rule, where it is found on the Infor CloudSuite forms, what numbers are allowed, and how it fits into the Scheduler's logic flow. Following that table is a figure which shows how a job progresses through the system from the Scheduler's rule point of view

Rule (Infor CloudSuite form)	Range of User Defined Rules Allowed	How the Rule Fits into the Scheduler Logic
Job Release Rule (Shop Floor Control Parameters - Scheduling tab [pre-SL 8.03] or Scheduling Parameters [SL 8.03 and later])	10-39	This rule is applied to provide a tie breaker to determine the sequence jobs are released into the system if two or more jobs have the same release date.
Sequencing Rule and Tiered Rules 1-3 (Resource form) and Global Sequencing Rule (Shop Floor Control Parameters – Scheduling tab [pre-SL 8.03] or Scheduling Parameters [SL 8.03 and later])	23-39	This rule applies when a job arrives at a resource group. The rule provides a ranking when a job enters the queue for a resource group.
Selection Rule (Resource form)	22-39	This rule applies when a resource becomes available to determine which job to process next at the resource.
Allocation Rule (Resource Groups)	8-39	This rule applies when resource is to be chosen from a resource group to process a job at an operation.

Rule (Infor CloudSuite form)	Range of User Defined Rules Allowed	How the Rule Fits into the Scheduler Logic
Setup Rule (Operations forms)	3-39	This rule is applied to determine whether a setup should be done at an operation.
Scheduler Rule and Setup Time Rule (Operations forms)	11-39	This rule is applied to determine the time a job spends on an operation.
Separation Rule (Batch Definitions)	3, 5-9	This rule is applied to determine how to group jobs together into batches, if not a batched production.
Release Rule (Batch Definitions)	4-9	This rule is applied to determine when to release a batch by determining what quantity a job applies to the batch, if not a batched production.
Override Rule (Batch Definitions)	2-9	This rule is applied to override the release quantity, if not a batched production.



Chapter 2: User Defined Rules

2

The Scheduler provides you with a wide variety of logic to use for modeling your manufacturing facility. This logic should cover the manufacturing system you will be modeling with this system. However, you might find that your facility contains some unique situations that require special logic to model them accurately.

This section contains the information you need to add this special logic to the Scheduler. You will learn how to add your own decision rules for ordering queues, batching loads, setup and run times, and when to setup .

The information in this section enables you to model an extended variety of situations in your manufacturing facility. This section provides a discussion of the procedures used to customize the rules within the Scheduler. Sections 3–6 provide descriptions of the functions, structures, and variables discussed in this section which support the construction of rules.

Reasons for Creating User Defined Rules

The standard Scheduler is intended to support the logic required to model most manufacturing operations. However, some facilities have unique manufacturing processes. To handle these unique situations, you can customize the Scheduler in a variety of ways. You can:

- Create new rules for sequencing and selecting queued requests for a resource.
- Create rules for selecting a resource from a resource group.
- Create new logic for forming and separating batches.
- Create extended ways to interpret the setup and run times.
- Create rules for determining when a setup should occur.

An example of such a situation is a heat treating operation which must be followed by a second heating operation within a given length of time or the product may crack. This implies that, before the first heating operation starts, the Scheduler may have to look ahead to see if the second heating can be done within the time limit. It may also require selecting loads for the second operation based on the criticality with respect to the time between operations. To customize the Scheduler for this situation, you would create a rule that would not allocate the furnace for the first operation unless the furnace for the second operation would be available within the time limit. You would then create a

resource selection rule for the second furnace to select the loads most critical with respect to the time limit.

Another example of special logic can be found in the processing of food and other agricultural products where availability of storage for intermediate products is a serious constraint. Before a product can begin processing, it may be necessary to determine if a downstream storage bin is available for the product. If a bin is available, it may be necessary to reserve that bin so competing groups of products cannot use it. In this case, you should customize a resource selection rule to perform this checking and reserving of storage.

Creating User Defined Rules

The first step in installing customized logic is to create a user code file which will contain your user written code representing your logic. You then use an editor to create the user code for your user-written rule. The code itself must be written in the C programming language. How this customized rule(s) fits into the phases of the Scheduler is discussed in Section 3.1.

After you have finished editing your logic in the editor, you are ready to compile and link your code. In the system this is done by choosing the *Make* option when you have finished editing user inserts in the Scheduler. This process is presented in Section 7. Once compiled and linked, your tailored logic is placed in a dynamic link library (dll) and is accessible to the Scheduler.

User Defined Rules

There are many instances in the Scheduler where you can choose from one of several decision rules listed in the drop down list and described in the online help messages. An example of this is a resource sequencing rule where you can choose from FIFO, LIFO, or several other rules to order requests as they are placed in the resource request queue. Notice that there are a number of rules listed as “user defined”. A user defined rule is a piece of customized logic that you can write to make decisions.

For every rule that you want to install, you must write one or more functions in C. To execute correctly, these functions must accept the proper arguments and return the proper value for the type of rule. The names of these functions must not conflict with the names of standard system functions; Section 3 describes the standard user-writable functions in detail, and the user-callable functions are described throughout this document, can be found in *factor.h* (see Section 3.2), and are listed in Section 6. The functions that you write are made accessible to the Scheduler by calls to installation functions in the initialization function *ucini1*.

The details of the rules that you can write are in the following subsections. The name of the installation function corresponding to each rule is given also. The details of installation and initialization function *ucini1* are given in Section 4.2.

A fragment from *ucini1* for installing a function for resource sequencing rule 39 is:

```
double sqrl39(LOAD*);

void ucini1()
{
     sedfrk(39, sqrl39);

     /* Install the rest of your first user-initialization
        customized logic here.  */
}
```

In the following subsections user-installable rule functions are shown. The names of these functions are only suggestions and do not have to be used. Following the suggested name, the suffix of “**” is used to represent a number of the rule that the function defines and should be replaced accordingly. An example is shown in each installation fragment where “**” is replaced by “39”.

Job Release Rules

Job release rules are used to determine the sequence of release for jobs with the same release date and time.

Your user-defined job release function can have any name that is not the name of a standard system function. It must accept a job(Type: ORDER*) as its only argument and return a value (Type: double) which is the ranking value of the job. Here is an example of an order release rule:

```
double orrl (ORDER *op)
/*-----------------------------------------------------------------
     Ranking function to cause new jobs to be ranked on a high to
          low priority.

     ARGS:
       op - pointer to job to evaluate ranking rule for

     RETURNS: job ranking value
-----------------------------------------------------------------*/
{
     return((double) -op->orprio);
}
```

To make your tailored job release function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *sedfok*. The function *sedfok* has two arguments:

1 The number of the job release rule for which your release function contains customized logic.

2 The address of your job release function.

Resource Sequencing Rules (Global Sequencing Rule)

Request queue sequencing rules are used to order queues of requests for resources as loads place the requests. To carry out this ordering, the logic of each sequencing rule is written into a load ranking function. For this reason, sequencing rules are often called load ranking rules. A load is often the job, but could be a subset of the job if offsetting or splitting is currently invoked or could be several jobs batched together. You can write and install custom load ranking functions to sequence queues by rules that are specific to your manufacturing operation.

Before creating your own load ranking functions, you should understand how requests are processed by the request queue sequencing rule. When a request is placed by a load, the request is given a ranking value by the load ranking function. The ranking value never changes. For example, suppose a sequencing rule ranks requests based on the time remaining until due date. At 1 p.m. today a load with a due date at 4 p.m. today places a request and is given a ranking value of 3. At 3 p.m. today, a load with due date at 5 p.m. today places a request and receives a ranking value of 2. Assume that these are the only requests in the queue. Suppose that one of these requests is satisfied at 3:30. The request that is satisfied is the one with ranking value of 2, even though it has 1.5 hours until it is due and the request with ranking value 3 has only 0.5 hour until it is due.

Requests are ranked in the internal list representing the queue from low to high based on the ranking value. You can reverse this order in your tailored logic by multiplying the ranking value by –1. For example, to rank loads based on a longest processing time for a current operation, a ranking function would return the negative of the processing time for the load at its current operation.

Your customized sequencing function can have any name that is not the name of a standard system function. It must accept two arguments in the following order:

1 A load (Type: LOAD*).

2 An attribute (Type: ATTRIBUTE).

It must return a value (Type: double) which is the ranking value of the load's request. Here is an example of a sequencing function to order loads based on least dynamic slack:

```
double sqrl (LOAD *ldp, ATTRIBUTE *atr)
/*---------------------------------------------------------------
  Ranking function to cause loads to be ranked on a least dynamic
  slack basis.

     ARGS:
       ldp - pointer to load for which to evaluate the ranking code

     RETURNS: load ranking value
---------------------------------------------------------------*/
{
     double rt, lt;
     int rn;

     rt = sermot(ldp, &rn, &lt);
     return(ldp->loordp->ordudt - DATENOW - rt);
```

```
}
```

To make your tailored load ranking function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *sedfrk*. The function *sedfrk* has two arguments in the following order:

1 The number of the sequencing rule for which your ranking function contains custom logic.

2 The address of your load ranking function

Resource Selection Rules

Request selection rules are used to order queues of requests for resources when requests are being removed from the queue. This occurs when a resource becomes available.

Before you create your own tailored selection rules and write the corresponding selection functions, you should understand the logic used by the selection function. When the following occurs: a resource becomes available, then the selection function is called. The request queues are contained in internal lists owned by the respective resource that are initially ordered by the sequencing rule. When the selection function is called, it reorders this list and returns the maximum number of requests to be considered in the allocation process. If the whole list is sorted, it returns the size of the list. Thus, if the number of requests returned is less than the size of the list, the sorted list for allocation may be only part of the queue of requests.

The allocation process begins by attempting to restart the operation for the first request on the sorted list for allocation that requires the number of available units or fewer. If the allocation process fails, the next request on the sorted list for the number of available units or fewer is considered. Successive requests on the sorted list for allocation are considered until the available units have been allocated or the end of the sorted list for allocation is reached. If there are available units after the last request is considered, they remain idle.

Selection functions differ from sequencing rules in that they may modify a request’s position in the request list each time that the resource becomes available. This allows requests to “age”. Consider the example from the preceding section where requests are both sequenced and selected by time until due date. At 1 p.m. today a load with a due date at 4 p.m. today places a request and is given a ranking value of 3. At 3 p.m. today, a load with due date at 5 p.m. today places a request and receives a ranking value of 2. Assume that these are the only requests in the queue. Suppose that a resource becomes available at 3:30. The request list would then be re-sorted with the request having ranking value of 3 first, because it has only 0.5 hours until it is due. The request with ranking value 2 would be second because it has 1.5 hour until it is due.

Your user defined selection function can have any name that is not the name of a standard system function. The selection function accepts as its only argument, a pointer to the resource (Type: RESRC*). All return a value (Type: int) which is the maximum number of requests to be processed. Here is an example of a resource selection function based on minimum setup:

```
int slrl (RESRC *rp)
/*-----------------------------------------------------------------
     Selection function to process a resource request list by using
```

```
     the minimum setup time for this resource on the first downstream
     setup or setup/operation operation for each load.  Loads which
     have no downstream setup or setup/operation operation are assumed
     to have a setup time of zero.

     NOTES:
     * The estimate flag is set to false for jscmsu so that it will
       use the current conditions to find the setup time.  Since the
       jobstep could be downstream, there is no guarantee that the
       current conditions for the resource in question will still
       be appropriate when the setup actually occurs.  Therefore,
       this rule should only be used when the structure of the model
       guarantees that the conditions for the downstream resource
       will be constant till the setup occurs.

     ARGS:
       rp - pointer to resource to resequence

     RETURNS: number of requests in the request list
-----------------------------------------------------------------------*/
{
     RREQ *rq;
     JOBSTEP *jsp;
     int i;

     for ( rq = (RREQ *) csfsls(rp->rsrqls);
           rq != NULL;
           rq = (RREQ *) csnxls((CSENTITY *) rq) )
     {

         /* Find first downstream setup jobstep.  */
         for ( jsp = rq->rrload->lojspt, i = 0;
               jsp != NULL && jsp->jstype != 4 &&
               jsp->jstype != 13 && i < 1000;
               jsp = jsp->jspnxt, i++);

         /* Compute setup time (zero if no setup jobstep).  */
         if (jsp == NULL || i >= 1000)
         {
             rq->rrprio = 0.0;

         }
         else
         {
            /* Place a tag for the resource.  This allows the step
               time to be computed properly for the downstream
               jobstep assuming that the load allocates the
```

```
                resource before any other load.  */

             dumrtag->rtrsrc = rp;
             cspols(rq->rrload->lorsls, (CSENTITY *) dumrtag, CSLIFO, NULL);
             rq->rrprio = jscmsu(0, rq->rrload, jsp);
             csgpls(rq->rrload->lorsls, (CSENTITY *) dumrtag);
          }
      }
      /*Sort list based on rankings.*/
      cssols (rp->rsrqls, serqor);

      /*return number in list*/
      return (csszls(rp->rsrqls));
}
```

To make your custom resource selection function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *sedfsl*. This function has two arguments in the following order:

1 The number of the selection rule for which your selection function contains custom logic.

2 The address of your selection function.

Notice that if you select a dynamic selection rule, the system will ignore your sequencing rule and use FIFO. If you write a “dynamic” selection rule, you should also emulate the system and force the use of FIFO. To do this, invoke function *setosq* on the applicable request list.

Resource Group Allocation Rules

Resource group allocation rules are used to decide which resources from a resource group will be allocated when allocation to a load is performed. They are also used for reallocation if reallocation has been enabled for the group.

You can create a custom resource group allocation rule and its corresponding function. The function can have any name that is not a standard system function name. It must accept four arguments in the following order:

1 A pointer to the resource group, Type: RESGRP *.

2 An integer for the number of resources to be selected from that group for allocation, Type: int.

3 A pointer to the load requesting to allocate from the resource group, Type: LOAD*.

4 A list to be populated with RMTAGS (group) of resources selected for allocation from the group, Type: CSLIST*.

Group members can be added and removed from the list of selected resources (*mlist*) using the system support functions *seadsl* and *sermsl*, respectively. To add group members to the list, use function *seadsl* as follows:

```
void seadsl(CSLIST *mlist, RESMEMB *rmp)
```

To remove group members from the list, use function *sermsl* as follows:

```
void sermsl(CSLIST *mlist, RMTAG *rmt)
```

Your custom function should return nothing (Type: void). Here is an example of a rule to select available resource group members on the basis of least mean utilization:

```
void rgsr (RESGRP *gp, int nu, LOAD *ldp, CSLIST *mlist)
/*-----------------------------------------------------------------
     Function to process resource group selection code 3, select
     required number of member resources that have the least mean
     utilizations (as a fraction of capacity).

     NOTES:
     * Ties are broken by selecting the first in the order listed in the
group.
     * This function requires that resource statistics be enabled.
       Any resource which does not have statistics enabled is
       assumed to have a utilization of zero.   Thus if all
       resource statistics are disabled, this degenerates to
       rule 0.
     * This function uses seaars, which includes must complete and
       maxOT in its considerations, to determine the availability
       of each member resource.

     ARGS:
       gp    - pointer to resource group to select member from
       nu    - number of units required from selected member
       ldp   - pointer to load to which resource will be allocated
       mlist - list to hold selected RESMEMB's

     RETURNS:  void
-----------------------------------------------------------------*/
{
     int i, j, *avail;
     double min, *util;
     RESMEMB *minp, *rmp;

     /* Allocate arrays for availability and utilization.  */
     avail = (int *) malloc(sizeof(int) * csszls(gp->rgrsls));
     if ( avail == NULL )
     {
         seferr(0, "Out of Memory");
     }
     util = (double *) malloc(sizeof(double) * csszls(gp->rgrsls));
     if ( util == NULL )
     {
         seferr(0, "Out of Memory");
```

```
    }

    /* Compute availability and utilization.  */
    for ( rmp = (RESMEMB *) csfsls(gp->rgrsls), j = 0;
          rmp != NULL;
          rmp = (RESMEMB *) csnxls((CSENTITY *) rmp), j++ )
    {

        avail[j] = seaars(rmp->rmres, ldp);
        util[j] = seutrs(rmp->rmres);
    }
    /* For the required number of units */
    for ( i = csszls(mlist); i < nu; i++ )
    {

        /* For each resource which is available.  */
        minp = NULL;
        for ( rmp = (RESMEMB *) csfsls(gp->rgrsls), j = 0;
              rmp != NULL;
              rmp = (RESMEMB *) csnxls((CSENTITY *) rmp), j++ )
        {

            if ( (! rmp->rmres->rsselfg) && (avail[j] > 0) )
            {

              /* Save if min.  */
                if ( (minp == NULL) || (util[j] < min) )
                {
                    minp = rmp;
                    min = util[j];
                }
            }
        }
        if ( minp == NULL )
        {
            break;
        }
        seadsl(mlist, minp);
    }
    free(avail);
    free(util);
    return;
}
```

To make your tailored resource group allocation function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *sedfgs*. The function *sedfgs* has two arguments in the following order:

1 The number of the resource group allocation rule for which your function contains tailored logic.

2 The address of your resource group allocation function.

Setup Rules (Determining Whether a Setup Is Necessary)

To determine whether a setup is necessary, you can write a function *ucwtsr* that will be called every time a non-standard system When-to-Setup rule is referenced (rule number greater than 2) and the standard step time rules 4 (Setup Lookup Table), 5 (Fixed Setup Time), 9 (Pre-Setup Table) or 10 (Pre-Setup Fixed) are called to compute the step time for the setup portion of the Operation. Function *ucwtsr* must have five arguments:

1 A pointer to a load, Type: LOAD*.

2 Type of resource (“R”esource), Type: char.

3 A pointer to a resource, Type: void*.

4 A pointer to an operation, Type: JOBSTEP*.

5 When-to-setup rule, Type: int.

It returns 1 for success and 0 for failure (Type: int).

The following example of *ucwtsr* sets up a resource every time the item and/or operation changes for the resources.

```
int ucwtsr(LOAD *ldp, char type, void *rp, JOBSTEP *jsp, int rule)
/*-----------------------------------------------------------------
     Function to set up resource if the item or operation is different from
last setup

     ARGS:
       ldp  - pointer to load
       type - "R"esource
       rp   - pointer to resource
       jsp  - pointer to operation
       rule - when-to-setup rule

     RETURNS
       true  - perform setup, or
       false - do not perform setup
-----------------------------------------------------------------*/
{
     int ireturn = 0
     char error[400];

     /* Check if setup, setup/operation, or super jobstep.    */
```

```
     if ( (jsp->jstype != 4) &&  (jsp->jstype != 13) &&  (jsp->jstype !=
19))
     {
         sprintf(error,"Jobstep not a Setup or Setup/Operate or Super
                    \n\nOrder ID  %s\nLoad ID  %d\nBatch ID
                    %ld\nJobstep ID  %s\nJobstep Type  %d\n",
                    ldp->loordp->orid, ldp->loid, (ldp->lobat == NULL)
                    ? OL : ldp->lobat->bibatid, jsp->jsid, jsp->jstype);
         seferr(0, error);
     }
     ireturn = (((RESRC *)rp)->rsptst != ldp->loordp->orptpt
                 ||   ((RESRC *)rp)->rsjsst != jsp) ? 1 : 0;
     return (ireturn);
}
```

It is not necessary to install *ucwtsr* in *ucini1* since it is called automatically when the setup rule is greater than 2 (i.e., a non-standard rule number).

Scheduler Rules (Run Time) and Setup Time Rules

Scheduler rules (Operation run time rules) and Setup Times Rules are used to determine the run/setup time for a load at a particular operation. Operation run/setup time functions can compute run/setup time based on the size of a load, the type of item the load represents, and other characteristics of a load. The run/setup time function can use the run/setup time entered on the operation and modify it as required by the run/setup time rule.

The run/setup time function is called under two circumstances:

- When a load begins processing at an operation for the first time.
- When an estimate of the run time for a load at a future operation is required, such as when computing dynamic slack.

The function is notified, through an argument, as to which of these situations apply. For many rules there will be no difference between the time calculations in either case. For rules which take into consideration the current situation at the operation such as setup, estimates are calculated differently. The run/setup time function is not called upon resuming a operation after a shift comes up. In this case, the run/setup time is the remainder of the previously computed run/setup time.

You can create custom operation run/setup time rules and their corresponding functions for your application. Your function can have any name that is not a standard system function name. It must accept three arguments in the following order:

1 An estimate flag (Type: int) set to:

   a TRUE (non-zero) for estimate

   b FALSE (zero) otherwise

2 A pointer to the load for which the estimate is being made, Type: LOAD*.

3 A pointer to the operation for which the time is to be computed, Type: JOBSTEP*.

4 An expression root, Type: void*.

Your function should return a value (Type: double), the total time for the operation in hours. Here is an example of a run time function to take a run time listed on a operation per piece that is entered on the operation:

```
double jsst (int est, LOAD *ldp, JOBSTEP *jsp, void *root)
/*-----------------------------------------------------------------
     Function to compute the run time for the given operation for
     a load using operation time code 1 (fixed time per part).

     ARGS:
       est - estimate flag
       ldp - pointer to load
       jsp - pointer to operation
       root - expression root
     RETURNS: the step time
-----------------------------------------------------------------*/
{
     double time;
     evalexp(root, 0, &time, 'R');
     return(time * ldp->losize);
}
```

To make your custom operation run time function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *jsdfst.* The function *jsdfst* has two arguments in the following order:

1 The number of the operation run time rule for which your function contains custom logic.

2 The address of your operation run time function

Batch Separation Rules

Batch separation rules are the part of a batch definition that specifies how arriving loads are to be separated into batch loads. The batch separation function associated with the rule is called for each load that arrives at a batching operation.

You can create a custom batch separation rule and its corresponding function. Your function can have any name that is not a standard system function name. The function must accept two arguments in the following order:

1 A pointer to the arriving load, Type: LOAD*.

2 A pointer to the batch definition specified on the operation, Type: BATCHDEF*.

Your function should return a pointer to a suitable batch load from the batch definition's list (Type: FORMBAT *) of forming loads or NULL if none exists. The forming batch load that your function selects should meet your requirements for separation and must not cause the forming batch quantity to exceed the maximum release quantity. If NULL is returned, the system will create a new forming batch load for the arriving load. Here is an example of a batch separation function:

```
FORMBAT *bsrl (LOAD *ldp, BATCHDEF *batch)
/*-----------------------------------------------------------------
     Function that processes the batch selection rule 1, which is
     to separate arriving loads into different batches with the same
     part number.
     ARGS:
       ldp   - pointer to load to find forming batch instance for
       batch - batch definition
     RETURNS:
       forming batch entity to put ldp into, or
          NULL if no appropriate batch load is forming
-----------------------------------------------------------------*/
{
     FORMBAT *fb;

     /* Look at each forming batch in the batch definition.  */
     for(fb = (FORMBAT *) csfsls(batch->btfmls);
         fb != NULL;
         fb = (FORMBAT *) csnxls((CSENTITY *) fb))
     {
         /* Return this forming batch if its children have the same
            part as the load passed in, and the new quantity will be
            less or equal to the maximum.  */
         if((strcmp(fb->fbldp->loordp->orptpt->panum,
                    ldp->loordp->orptpt->panum) == 0) &&
            (fb->fbquant + sequfb(ldp, batch, batch->btqurl)
            <= batch->btmax))
         {
            return(fb);
         }
     }

     /* return null if we couldn't find one.  */
     return(NULL);
}
```

To make your customized batch separation function available to the Scheduler, you must "install" it from the first user-initialization function *ucini1* by calling the function *sedfbs*. The function *sedfbs* has two arguments in the following order:

1 The number of the batch separation rule for which your function contains tailored logic.

2 The address of your batch separation function

Batch Release Rules

Batch release rules are the part of a batch definition that specifies by what amount an arriving load increases the forming batch quantity of the batch load to which it is to be accumulated. A batch release function is called by a batch separation function to determine if there is room in a forming batch load for the arriving load. Once a forming batch load has been selected for the arriving load, the batch release function is called again in order to increment the forming batch quantity.

You can create a custom batch release rule and its corresponding function. Your function can have any name that is not a standard system function name. It must accept two arguments in the following order:

1 A pointer to the arriving load, Type: LOAD*.

2 A pointer to the batch definition specified on the operation, Type: BATCHDEF*.

Your function should return the amount by which the forming batch load should be incremented (Type: double). Here is an example of a batch release function:

```
double bqrl (LOAD *ldp, BATCHDEF *batch)
/*-----------------------------------------------------------------
     Function to process quantity rule 1 (1 per part on load).

     ARGS:
       ldp   - pointer to load being added to batch
       batch - batch definition

     RETURNS: load size of ldp
-----------------------------------------------------------------*/
{
     return((double) ldp->losize);
}
```

To make your custom batch quantity function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *sedfbq*. The function *sedfbq* has two arguments in the following order:

1 The number of the batch release rule for which your function contains tailored logic.

2 The address of your batch release function.

Batch Override Rules

Batch override rules are used to determine if forming batch loads should be released even though the forming batch quantity has not reached the minimum release quantity. An example of such a situation is when a load has been waiting for a length of time longer than that specified in the override release threshold on the batch.

The batch override function associated with a rule is called under these two conditions:

- When add override release reviews are enabled and a load is added to a forming batch, but the forming batch does not reach or exceed its minimum release quantity.
- During a periodic override release review, which occurs at an interval specified on the batch if the periodic review is enabled.

You can create a custom batch override rule and its corresponding function. Your function can have any name that is not a standard system function name. It must accept two arguments:

1 A pointer to the forming batch load, Type: FORMBAT *.

2 A pointer to the batch for the forming batch load, Type: BATCHDEF *.

Your function should return (Type: int):

- TRUE (non-zero) if the forming batch load should be released.
- FALSE (zero) if the forming batch load that should not be released.

Here is an example of a batch override function:

```
int borl (FORMBAT *formbat, BATCHDEF *batch)
/*-----------------------------------------------------------------
     Function to process batching override rule 0 (the time that
     the forming batch has been waiting is greater or equal to the
     override threshold).

     ARGS:
       formbat - forming batch load to check for release
       batch   - batch that the forming batch is in

     RETURNS:
       true if the forming batch has been waiting too long, or
       false otherwise.
-----------------------------------------------------------------*/
{
     double x;
     char trace[200];

     x = cstnow - formbat->fbsttim;
     sprintf(trace, "Batch %d has been forming %f hours",
             formbat->fbldp->lobat->bibatid, x);
     setrace(0, trace);
```

```
     /* If the time the batch has been forming is at least the
        threshold, then release the batch.  */
     if(x >= batch->btovth)
     {
         return(1);
     }
     else
     {
         return(0);
     }
}
```

To make your tailored batch override function available to the Scheduler, you must “install” it from the first user-initialization function *ucini1* by calling the function *sedfov.* The function *sedfov* has two arguments:

1 The number of the batch override release rule for which your function contains tailored logic.

2 The address of your batch override function.

Example

This example is a resource selection rule that will select the next job based on the standard resource selection number 17, which is Changeover Processing Time/Long Wait. When the standard rule 17 is called it will rank candidates in one of two ways: 1) if there will not be a setup, then jobs are ranked by setup time, then by waiting time, or 2) if there will be a setup, then jobs are ranked by waiting time for all jobs that will change the setup. The problem with the standard rule is that there is no secondary ranking rule, in other words, ties in waiting time are not broken in any explainable way. This rule will change that processing slightly by breaking ties in waiting time using the resource’s sequencing rule. This processing can now be done with the standard product using the Tiered Selection Rule.

Below is a description of each function in the code:

ucini1

Install the rule function, *rsel_25*, as rule number 25.

ucini2

Set the type of any resource that uses rule 25 to 3. This will cause the data to be collected for cumulative processing time. Other choices for the type are 0 (most standard rules), 1 (number of jobs – used by standard rules 12 and 13), 2 (number of items – used by standard rules 14 and 15), 3

(accumulated processing time - used by standard rules 16 and 17), and 4 (elapsed time – used by standard rules 18 and 19).

rsel_25

- Set pCurRes as pointer to the resource, which will be used by a function later.
- Call the standard rule 17 (function
- *serqsl*) which will create a sequenced list of candidate jobs.
- Set rprio1 as the priority value of the first job in the prioritized list.
- If the number of candidates (numreq) is equal to the number in the queue and the priority of the first job is not zero (setup time), then we are changing setups and the list is currently ranked by waiting time with all non-zero setups at the top of the list. In this case, ties in waiting time will be broken by the sequencing rule. The function
- *cssols* reorders the list with the function *rqor25seq* being a comparison function for a pair of jobs on the list. *cssols* goes through the list comparing jobs by twos until an ordered list by waiting time and sequencing rule is achieved.
- Else if the number of candidates is greater than zero, then we are not changing setups. In this case, ties in setup time will be broken by waiting time, while ties in waiting time (within the same setup time) will be broken by the sequencing rule. The function
- *cssols* reorders the list with the function *rqor25wt* being a comparison function for a pair of jobs on the list. *cssols* goes through the list comparing jobs by twos until an ordered list by setup time (either positive or zero), waiting time, and sequencing rule is achieved.
- Else if the number of candidates is zero, then we have not yet reached the desired number of setups and a Setup Delay is being scheduled.
- Function
- *rsel_25* returns the number of jobs in priority order to be considered for allocation.

rqor25seq

- This function compares two jobs based on priority (waiting time) and if there is a tie, attempts to break the tie based on the sequencing rule.
- Return values are: -1: request 1 has higher priority, 1: request 2 has higher priority, and 0: requests are equal priority.

rqor25wt

- This function compares two jobs based on priority (setup time), and if there is a tie, attempts to break the tie based on waiting time, and if there is still a tie, attempts to break the tie based on the sequencing rule.

- Return values are: -1: request 1 has higher priority, 1: request 2 has higher priority, and 0: requests are equal priority.

Below is the actual user code representing the case described above:

```
#include "factor.h"

RESRC *pCurRes;

static int rsel_25(RESRC *rp);
static int rqor25seq(CSENTITY *ep1, CSENTITY *ep2);
static int rqor25wt(CSENTITY *ep1, CSENTITY *ep2);

void ucini1(void)
{
    sedfsl(25, rsel_25);
}

void ucini2(void)
{
    RESRC *pstRes;

    /* set selection rule type for user-written rules */
    for (pstRes = (RESRC *) csfsls(ssgvar.sgresrc); pstRes != NULL;
         pstRes = (RESRC *) csnxls((CSENTITY *) pstRes))
    {
        if (pstRes->rsslrl == 25)
        {
           pstRes->rssltype = 3;
        }
    }
}

int rsel_25 (RESRC *rp)
/*  Resource selection rule 25 for Changeover Processing Time/Longest
    Wait/Sequence Rule */
{
    int numreq;
    RREQ *rq;
    double rprio1 = 0.0;

    /* Set resource pointer for later use */
    pCurRes = rp;

    /* Call standard rule 17 (Changeover P.T./Longest Wait) */
    numreq = serqsl(rp, 17);
```

```
    /* Get priority of first load waiting */
    rq = (RREQ *) csfsls(rp->rsrqls);
    if (rq != NULL)
    {
        rprio1 = rq->rrprio;
    }

    /* There is going to be a change so use sequencing rule as secondary
       ranking, i.e. considering all requests and first is non-zero setup
       time */
    if (numreq == csszls((CSENTITY *) rp->rsrqls) && rprio1 > 0.0)
    {
        /* Sort list based on changover then sequencing rule. */
        cssols(rp->rsrqls, rqor25seq);
    }

    /* Need to rank zero setups by Waiting Time, then sequencing rule */
    else if (numreq > 0)
    {
        cssols(rp->rsrqls, rqor25wt);
    }

    return(numreq);
}

static int rqor25seq(CSENTITY *ep1, CSENTITY *ep2)
/*---------------------------------------------------------------------
        Ordering (comparison) function to return the relationship of
        resource requests based on the following tiered criteria:

             Current priority
             Sequencing rule value

        Args:
         ep1 - pointer to resource request 1
         ep2 - pointer to resource request 2

        Returns:
         -1 - request 1 should go before request 2 in list
          0 - request 1 has same priority as request 2
          1 - request 1 should go after request 2 in list

---------------------------------------------------------------------*/
{
        RREQ *rq1, *rq2;
        double rv1, rv2;
```

```
        rq1 = (RREQ *) ep1;
        rq2 = (RREQ *) ep2;

        /* Return if order priorities are not equal */
        if (rq1->rrprio < rq2->rrprio)
           return(-1);
        if (rq1->rrprio > rq2->rrprio)
           return(1);

        /* If still tied break based on sequencing rule. */
        rv1 = selork(rq1->rrload, pCurRes->rssqrl, NULL);
        rv2 = selork(rq2->rrload, pCurRes->rssqrl, NULL);

        if (rv1 < rv2)
           return(-1);
        if (rv1 > rv2)
           return(1);
        return(0);
}



static int rqor25wt(CSENTITY *ep1, CSENTITY *ep2)
/*---------------------------------------------------------------------
        Ordering (comparison) function to return the relationship of
        resource requests based on the following tiered criteria:
             Current priority
             Sequencing rule value

        Args:
         ep1 - pointer to resource request 1
         ep2 - pointer to resource request 2

        Returns:
         -1 - request 1 should go before request 2 in list
          0 - request 1 has same priority as request 2
          1 - request 1 should go after request 2 in list

---------------------------------------------------------------------*/
{
        RREQ *rq1, *rq2;
        double rv1 = 0.0, rv2 = 0.0;
        CSENTHD *hp1, *hp2;

        rq1 = (RREQ *) ep1;
```

```
        rq2 = (RREQ *) ep2;

        /* Return if order priorities are not equal */

        if (rq1->rrprio < rq2->rrprio)
           return(-1);
        if (rq1->rrprio > rq2->rrprio)
           return(1);

        /* If tied break based on waiting time. */
        hp1 = (CSENTHD *)rq1;
        --hp1;
        rv1 = hp1->_letime;
        hp2 = (CSENTHD *)rq2;
        --hp2;
        rv2 = hp2->_letime;

        if (rv1 < rv2)
           return(-1);
        if (rv1 > rv2)
           return(1);

        /* If still tied break based on sequencing rule. */
        rv1 = selork(rq1->rrload, pCurRes->rssqrl, NULL);
        rv2 = selork(rq2->rrload, pCurRes->rssqrl, NULL);

        if (rv1 < rv2)
           return(-1);
        if (rv1 > rv2)
           return(1);

        /* still tied return 0 */
        return(0);
}
```

Chapter 3: Internal Data Structures and Inner Workings

3

This section is devoted to defining the scheduling process, the internal data structures, details of the event system, how to manipulate lists (for example, the list of jobs waiting to be processed by a resource), initialization functions for the scheduling run, and creating error, warning, and trace messages. These are all of the internal structures and functions one needs to support creating rules.

Scheduling Process

To write effective user code, you must understand the purpose of the functions that you can write and the point in the scheduling process at which they are called. A summary of the scheduling process for a system model is as follows:

Pre-initialization

When you choose **Schedule** from the Scheduling form, the alternative row from the database table ALTSCHED for the alternative is read, and the members of the global structure *ssgvar* are initialized.

First User Initialization

This part of the Scheduler is executed by the optional user-writable function *ucini1*. The purpose of this function is to install user-written rule functions and to define and initialize any data structures required by the rest of the user-written portion of the scheduling model. Any additional functionality is discouraged.

Initialization

The system model for the alternative to be scheduled is loaded into memory and initialized.

Second User Initialization

This part of the scheduler is executed by the optional user-writable function *ucini2*. The purpose of this function is to perform any initialization of the user-defined data structure which is dependent on the scheduler data. Most auxiliary data is attached to objects here.

Scheduler Execution

Once all the data is loaded and initialized the Scheduler is executed. All requested raw output data is written to the database, and all requested statistics are maintained. User-written rule functions are called during the scheduler execution. Functions associated with the creation of new entities are called during the Scheduler.

Save-First User Finalization Function

This part of the Scheduler is executed by the optional user-writable function *ucfin1*. The purpose of this function is to enable you to adjust any of the system statistics to reflect any discrepancies created by the user-written portion of the model. It is very seldom used.

Save-Summary Statistics Storage

All summary statistics which were requested will be written to the database.

Exit

The Scheduler session is terminated. The user-writable function *ucend* is called during this phase so that you can close open files or perform any tasks desirable at that point.

Internal Data Structures

When you write user defined rules to implement processing logic specific to your application, you will need to be familiar with the internal data structures of the Scheduler. This section contains the header files for you to use as a road map of these internal structures. By studying these header files, “factor.h” and “fproro.h”, for example, you can determine how operation processing communicates with the resource selection process. The internal data structures are similar to the system database structure. In many cases, the tables and columns in the database map directly to the structures and variables in the internal data structures.

The header file “factor.h” contains most of the system component data structures and other constants, includes, and so on, required by the Scheduler. This file is installed in your <Directory>\APS\Scheduler\UserCode folder, where <Directory> is the product’s installation folder (for example, Infor). It is a text file which can be printed or viewed using any text editor or word processor.

The header file “fproto.h” contains all the function prototypes needed by the user for functions defined in Sections 2 through 6 of this manual. This file is installed in your <Directory>\APS\Scheduler\UserCode folder, where <Directory> is the product’s installation folder (for example, Infor). It is a text file which can be printed or viewed using any text editor or word processor.

Organization of Internal Data Structures

The internal data structures serve as a link between the standard Scheduler and the code that you write to customize the standard Scheduler. The internal data structure definitions are contained in the header file “factor.h” (see previous section). For this reason, you must include the statement

```
#include "factor.h"
```

at the beginning of the code for your Scheduler customization. Once you have included this line in your code, you can use the data defined in “factor.h” in your functions.

The data defined in factor.h falls into three categories:

- Modeling objects
- Internal objects
- Global scheduling structures

The modeling objects are closely related to the input database tables and their columns. You can see this by comparing factor.h to the input database files which end in “000” in the database. For example, the table RESRC000 contains the data for resources.

Each modeling object contains a pointer for auxiliary data to be attached. Auxiliary data can be thought of as user data to be used by the Scheduler for an object. The auxiliary data pointer is a (void *) so you can attach anything to the object, i.e. pointer to a structure, integer value, a string, etc. For example, the resource structure, RESRC, contains an element *(void *) rsaux*. Auxiliary data would most likely be populated in *ucini2*.

The internal objects are used in queuing and other internal Scheduler processing. Internal objects are often closely related to modeling objects and may even be pointers to modeling objects. A resource request is an example of an internal object that is a pointer to a modeling object, a resource. Internal objects are used instead of modeling objects on queue lists and other lists. This is done because entities can be on at most one list and many modeling objects are on a global list. Restricting entities to one list only helps to preserve data integrity and expedites searching and processing.

The global structures include data related to the alternative table in the database, the problem definition table in the database, the alternative summary table in the database, lists of members of the structures corresponding to the modeling components, and structures used in the internal workings of the Scheduler.

Events and the Event Calendar

If the custom logic that you need to install does not fit into one of the user-installable rules discussed in Section 2.3., you can write a function for your own event. You may also need to access the event calendar or schedule an event from your custom rule. An example where you might want to do this is periodic inventory valuation.

Events are calls to functions that occur at scheduled times. Events are scheduled by adding them to an *event list*. Two event lists, also called *event calendars*, are used by the Scheduler to store events. These event lists are defined as follows:

- standard - This list contains standard and custom events in the order in which they are scheduled to occur. You can schedule your custom events on this list.
- internal - This list contains events which happen immediately. All of the events on this list are processed before any event on the standard list is processed.

The following special system functions can be used to schedule events and to manipulate event lists in other ways.

Function	Description
cschd0	Schedule a zero-time system event (on the internal list).
csched	Schedule a timed system event (on the standard list).
cselep	Search the event lists for an entity.
cselfv	Search the event list for the first entity scheduled for a given function.
cselnv	Search the event list for the next entity scheduled for a given function.

Function	Description
csepea	Get the scheduled event address for an entity.
csepet	Get the scheduled event time for an entity.
csnew	Get a pointer to a new entity of specified size.
CSNEW	Get a pointer to a new entity of specified size. (The difference between *CSNEW* and *csnew* is discussed in Section 3.4.2).
csterm	Terminate an entity pointer.
uccschd0	Schedule a zero-time user event (on the internal list).
uccsched	Schedule a timed user event (on the standard list).

You should not use the standard list functions described in Section 3.4 to manipulate event lists because event lists contain many different types of entities and their ordering is more complex than other lists.

Your custom event function must accept only one argument, a pointer to a CSENTITY. A CSENTITY must be allocated using function *csnew*, but the contents of the entity can be anything you like. For example, your custom event function should have the form:

```
void evntfun(CSENTITY *)
{
     /* Tailored event logic.  */
}
```

and could be scheduled to occur one hour from the current Scheduler time with a call to the function *uccsched:*

```
uccsched(ep, "EVNTFUN", 1.0);
```

You must schedule the first occurrence of your custom event from *ucini2* or from some custom functions that you have written. You can schedule subsequent events from the event function itself or from custom functions you have written.

List Manipulation

Much of the internal system data is contained in lists. For this reason, the custom logic that you write will probably involve a considerable amount of list manipulation. To expedite the list manipulation in

your custom logic, the system provides you with a number of list manipulation functions that you can call from your code.

Many of the internal system lists contain pointers to scheduler constructs which are also called *entities*. Event lists, discussed in Section 3.3, contain entity pointers for individual events. An entity pointer is the argument passed to the event function itself. Many system functions return pointers to entities. Many of your custom functions will return entity pointers.

You can use system functions to add entities to lists, remove entities from lists, reorder lists, create lists, and traverse lists. These functions are generic in nature, exploiting the fact that passing pointers requires no knowledge about the attributes of the entities they represent.

Creating Lists

You can create lists within, and for use with, the internal data structures you create for the Scheduler. A list is essentially a group of entities maintained in a particular order. The order is defined when the list is created, but can be overridden for particular operations on a list. You can use a list as a holding place for an indeterminate number of entities (a boundless array) or to represent a queue within the scheduling model.

You must access a list through the use of a list pointer (pointer to a list), which you can obtain from one of these two places:

- The function *csmkls* to make a list with statistics.
- The macro *CSMXLS* to make a list without statistics.

Note that list pointers are not entity pointers, so lists cannot be placed in other lists. However, an entity data structure can contain one or more list pointers, and that entity can then be placed in a list.

You can choose a FIFO or a LIFO ordering for your list, or you can install a custom function to order your list. You can do this through the arguments on the function *csmkls* and the macro *CSMXLS*, as documented in the header file *factor.h* Observe that the arguments of both *csmkls* and *CSMXLS* include an ordering type and a pointer to an ordering function. The ordering type may be one of the following:

Type	Description
CSFIFO	The list is maintained in first-in-first-out (FIFO) order. New entities are inserted at the end of the list.
CSLIFO	The list is maintained in last-in-first-out (LIFO) order. New entities are inserted at the beginning of the list.

Type	Description
CSORDF	The list is ordered based on a user-defined ordering function, with FIFO ordering used in the case of ties.
CSORDL	The list is ordered based on a user-defined ordering function, with LIFO ordering used in the case of ties.

For CSFIFO and CSLIFO, you need not install an ordering function and you must use NULL for the pointer to the ordering function. For CSORDF and CSORDL, you should use the pointer to your custom ordering function for this argument.

List ordering functions provide a criterion for comparing two entities on the list. The system inserts entities based on the criterion given in the function. The list ordering function is called automatically by the system each time a new entity is inserted in the list.

Your custom list ordering function must accept two arguments, both of which are pointers to the entities of the type which the list is to hold. Your function should obtain a value for each of the two entities based on some attribute of the entity. Your function should return an integer indicating which entity should be placed ahead of the other in the list as follows:

- If entity1 should be placed ahead of entity2, then return a value < 0.
- If entity1 should be placed after of entity2, then return a value > 0.
- If the entity1 and entity2 are tied, then return 0.

The list is maintained in ascending order based on the criterion in the ordering function. To provide a descending ordering, simply reverse the sign of the return values (i.e., multiply by –1).

The following example is a function to order a list first in alphabetical order based on job ID and second in numerical order based on load ID:

```
int compfun(lt1, lt2)
LOADTAG *lt1, *lt2;
{
     int ord;

     /* Check whether jobs are identical.  */
     ord = strcmp(lt1->ldp->loordp->orid,
                  lt2->ldp->loordp->orid);

     /* If not identical, return the ordering given by strcmp.  */
     if (ord != 0)
     {
         return(ord);
     }
```

```
    /* If identical, return the ordering by load.  */
    return(lt1->ldp->loid - lt2->ldp->loid);
}
```

The relationship returned by the standard C function *strcmp* has the same convention as the list ordering function. If the orders are not identical, the comparison of the IDs may be returned as is. If the orders are identical, further comparison must be made on the basis of their load IDs. Since the load IDs are integer, subtracting the second load ID from the first results in the required comparison return value. The ordering code on a list can be changed using the function *setosq*. This is useful for request lists that will be reordered by the selection rule, i.e., Dynamic Slack. The list ordering can be changed to CSFIFO which eliminates searching when putting entities into the list.

Creating Entities for Lists

To create entities with which you can populate lists, you can use the function *csnew* or the *CSNEW* macro. The function *csnew* takes the size of an entity as an argument and returns a pointer to a newly created entity which is going to be placed in a list. The size argument is normally expressed as a “sizeof” construct involving the “typedef” for the entity. For example, a code fragment to create a request entity would be:

```
RREQ *rqp;

rqp = (RREQ *)csnew(sizeof(RREQ));
```

This type of statement was used so often that the *CSNEW* macro was created. Using this macro, the above fragment becomes:

```
RREQ *rqp;

rqp = CSNEW(RREQ);
```

While this macro takes care of the multiple references to the type name in this case, casting is still a way of life when dealing with entity pointers.

You must create all entities for use with list or event functions using *CSNEW* or *csnew*. You should note that while the standard C runtime library also uses character pointers to return an allocated block of memory, such a block of memory cannot be used with the list functions. Furthermore, the address of a static or automatic variable cannot be used with these functions either.

Traversing Lists

You can step through lists one entity at a time from one end to another. You would do this when populating auxiliary attributes of internal data structures, searching lists of requests for the “best” candidate according to a selection rule, and many other situations.

You can traverse a list using a “for” loop and the following system functions:

- *csfsls* - Returns a pointer to the first entry.
- *csnxls* - Returns a pointer to the next entry.
- *cslsls* - Returns a pointer to the last entry.
- *csprls* - Returns a pointer to the previous entry.

These traversal functions return a NULL pointer to indicate the end of the list. For example, consider the following code to traverse the global list of resources:

```
RESRC *rp;

for (rp = (RESRC *)csfsls(SSGVAR.sgresrc);
     rp != NULL;
     rp = (RESRC *)csnxls(rp))
{

     /* Processing for each resource in the list.  */
}
```

You can modify the “for” loop to terminate the search when a specific material was located, either by adding an additional “and” clause to the continuation expression or by adding an “if” with a “break” inside the body of the loop. The following example illustrates traversing a list in reverse order and terminating either at the end of the list or at the first resource with an infinite flag equal to 1, i.e. infinite resource.

```
RESRC *rp, *prev_rp;

for (rp = (RESRC *)cslsls(SSGVAR.sgresrc);
     rp != NULL && rp->rsinffg == 1;
     rp = prev_rp)
{

     prev_rp = (RESRC *)csprls(CSENTITY *rp):

     /* Processing for each resource in the list.  */
   }
```

Note that the pointer to the next or previous entity in the list is always accessed from the pointer to the current entity. A serious problem would result if the current entity were removed from the list prior to getting the pointer to the next entity. In the previous example, this problem was solved by retrieving the pointer to the entity prior to the current entity before the processing within the loop. Thus, the processing may remove the current entity from the list if desired.

Inserting and Removing Entities in Lists

Several functions are provided for inserting and removing entities in lists. You cannot have an entity in more than one list at a time. You can insert and remove entities into lists using the functions in the table below.

Function	Description
csptls	Insert the entity into the list at the proper position for the ordering defined on the list.
cspols	Put the entity on the list using a one-time ordering, as when putting an entity at the head of the list regardless of the list ordering.
csppls	Insert the entity prior to a certain entity already in the list.
cspfls	Insert the entity following a certain entity already in the list.
csgtls	Remove by an entity's position in the list.
csgpls	Remove by a pointer to the entity.

The following example illustrates the way entities are inserted and removed from a list. This function builds a list of materials which have a quantity on hand below some threshold. It then passes the list to a function named "process" to do some processing of the information in the list. Finally, the function empties and deletes the list.

```
typedef struct
{
     MATL *mp;      /* Pointer to a material.  */
}UCMATL;

#define UCTHOLD 100.0

void lowqoh()
{
     MATL       *mp;
     CSLIST     *lp;
     UCMATL     *ucp, *nucp;

     lp = CSMXLS("", CSFIFO, NULL);

     for ( mp = (MATL *)csfsls(SSGVAR.sgmatl);
```

```
            mp != NULL;
            mp = (MATL *)csnxls(CSENTITY *mp))
      {

         if ( mp->mtqoh <= UCTHOLD )
         {
                ucp = CSNEW(UCMATL);
                ucp->mp = mp;
                csptls(lp, CSENTITY *ucp);
         }
      }
      process(lp);

      for ( ucp = (UCMATL *)csfsls(lp);
            ucp != NULL;
            ucp = nucp)
      {
            nucp = (UCMATL *)csnxls(CSENTITY *ucp);
            csgpls(lp,ucp);
            csterm(ucp);
      }
      csdlls(lp);
}
```

Reordering Lists

Lists can be reordered in the following two ways. The method you choose depends on the situation.

- Reorder manually using the functions described in Section 6.6 to remove entities from their current position and place them in their desired position.
- Reorder by calling function *cssols* to sort the list using the given ordering function.

When you order lists manually, you should realize that the functions to remove entities from lists and the functions to add entities to lists change the statistics of the list. This is usually not a problem for lists that are used as a holding place for an indeterminate number of entities since statistics are usually not needed or collected for these lists. It *is* a problem for lists representing queues because queue statistics can be useful for analysis and decision making. Statistics are almost always kept for lists representing queues. When manually ordering lists you should temporarily disable statistics. You can use the functions in the table below to disable and enable statistics.

Function	Description
csdsst	Disable statistics collection.
csenst	Enable statistics collection for a list.

You need not disable or enable statistics collection when using *cssols* to reorder a list because *cssols* automatically maintains statistics.

An example of manually reordering a list is the selection rule given below that moves the first acceptable request to the top of the list. The function loops through the request list until either an acceptable request is found or the end of the list is reached. If the end of the list is reached and no suitable request is found, the function returns a zero. If a request is found, the statistics are disabled using function *csdsst*, the acceptable request is removed from the list using function *csgpls* to remove it by pointer, and then it is inserted at the head of the list using function *cspols* with a one time ordering of LIFO. Finally, the statistics are enabled using function *csenst* and a value of one returned to indicate successful completion.

```
int selfun(RESRC *rp)
{
     RREQ *rqp;
     int accept();
     CSOBSST *os;
     CSTIMST *ts;

     for (rqp = (RREQ *)csfsls(rp->rsrqls));
          rqp != NULL && (!accept(rqp));
          rqp = (RREQ *)csnxls(CSENTITY* rqp);

     /* If no acceptable request is found...  */

     if (rqp == NULL)
     {
         return(0);
     }
     csdsst(rp->rsrqls, &ts, &os);
     csgpls(rp->rsrpls, rqp);
     cspols(rprsqls, rqp, CSLIFO, NULL);
     csenst(rp->rsrqls, ts, os);
     return(1);
}
```

If you had wanted to sort the whole list rather than move one entity to the top, it would have been much easier to use the function *cssols*. Sorting a whole list manually would require you to implement some type of sorting algorithm.

Initialization Functions

The user-installable rules must be made available to the Scheduler. The system provides a number of user-installable points for initialization of data and installation of rules. You can create system

functions to initialize the Scheduler. The function *ucini1* is called after global variables are read and is where custom rules are installed.

Error and Warning Messages

To report fatal errors during the scheduling run, you can use the system function *seferr*, and to report non-fatal errors (warnings) during the Scheduler, you can use the system function *sewarn*. To report input errors and warnings, you can use the system functions *sierr* and *siwarn*, respectively.

Internally to the system, the functions accept a variable number of arguments (similar to *printf*) to be used in the error message. The first argument must be an integer for the number of the error message to write. For the remaining arguments, the system insures that the numbers and types match the format specified.

Your calls to these functions should use two arguments in the following order:

1 A 0 for user-defined error messages.

2 A string for the error message to write. *Note:* You can use *sprintf* to create the string to pass to the function.

For examples of using these functions, see the example in Section 2.4.

Trace Messages

To write trace messages to the trace file and/or window, you can use the system function *setrace*.

Internally to the system, the function accepts a variable number of arguments (similar to *printf*) to be used in the trace message. The first argument must be an integer trace number for the number of the trace message to write. For the remaining arguments, the system insures that the numbers and types match the format specifiers.

Your calls to this function should use two arguments in the following order:

1 A 0 for user-defined error messages.

2 A string for the trace message to write. *Note:* You can use *sprintf* to create the string to pass to the function.

For examples of using *setrace*, see the example in Section 2.4.

Chapter 4: User-Writable Functions

4

This section presents the two categories of user-writable functions: scheduling rules and scheduling rule support functions. Within each category, these functions are presented in alphabetical order. For each function, the following information is provided:

- A short description.
- A synopsis of the function type and, as applicable, its arguments and header files.
- The value(s) returned by the function (as applicable).
- Programming notes (as necessary).

When writing a decision rule, you may use the user-callable support functions described in Section 6. The user-writable functions are presented in the following subsections:

- User-Writable Scheduling Rules
- User-Writable Scheduling Rule Support Functions

User-Writable Scheduling Rules

This section presents the user-writable decision rules. You can write these rules to supplement the list of standard rules supplied by the system. The rules are presented in alphabetical order. For assistance in writing and installing a decision rule, see Section 2.3.

Job Release Rule

Description

Job release rules are called to rank jobs with identical release dates.

Synopsis

```
#include "factor.h"

double myrule(op)

ORDER *op;        /* Pointer to the order. */
```

Returns

A ranking value for the job.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfok (39, myrule);
```

Note: The order with the lowest ranking value precedes all other orders with the same release date in the list of orders. See section 2.3.1 for further discussion of this rule. Function Siorrk may be used to call standard rules.

Resource Sequencing Rule

Description

Request queue sequencing rules are called to determine a ranking value for the load.

Synopsis

```
#include "factor.h"

double myrule(ldp, atrib)

LOAD      *ldp;  /* Pointer to the load. */
ATTRIBUTE *atr;  /* Pointer to attribute, if any, to base ranking on. */
```

Returns

A ranking value for the load.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfrk (39, myrule);
```

Note: Request queue sequencing rules are called to rank the loads waiting in resource queues. The load with the lowest ranking value is first in the list. See Section 2.3.2 for further discussion of this rule. Function *selork* may be used to call standard system rules.

Resource Selection Rule

Description

The resource request selection rule specified on a resource is called to order the request list before a request is selected to be satisfied.

Synopsis

```
#include "factor.h"

int myrule(rp)

RESRC *rp;        /* Pointer to the resource. */
```

Returns

The maximum number of requests to consider.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfsl (39, myrule);
```

Note: See Section 2.3.3 for further discussion of this rule. Function *serqsl* may be used to call standard system rules.

Resource Group Allocation Rule

Description

The group member allocation rule specified on a resource group is called to select one or more members from the group.

Synopsis

```
#include "factor.h"

void myrule(gp, nu, ldp, mlist)

RESGRP  *gp;      /* Pointer to resource group. */
int      nu;      /* Number of units required.  */
LOAD    *ldp;     /* Pointer to the load.       */
CSLIST  *mlist;   /* Current selection list.    */
```

Returns

Nothing.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfgs (39, myrule);
```

Note: If the selection list does not contain the required number of selections when the rule returns, the selection has failed.

Note: The functions *seadsl* and *sermsl* should be used to add and remove selections, respectively.

Note: See Section 2.3.4 for further discussion of this rule.

Note: Function *sergms* and *seslrg* may be called from a member selection and allocation rule, respectively, to call standard system rules.

Setup Rule (When to Setup)

Description

The setup rule specified on an operation determines whether a setup is required.

Synopsis

```
include "factor.h"

int ucwtsr (ldp, rp, jsp, rule)

LOAD    *ldp;   /* Pointer to the load.               */
char     type;  /* 'R'esource                         */
void    *rp;    /* Pointer to the resource to setup.  */
JOBSTEP *jsp;   /* Pointer to the operation.          */
int      rule;  /* When-to-setup rule to use.         */
```

Returns

True if setup is to be done, False if no setup is to be done.

Install

None.

Note: This rule is not installed, instead the function *ucwtsr* is always called when the rule number is greater than 2 (i.e., non-standard rule).

Note: This rule may be called several times before the setup actually occurs. Therefore, you should not write your rule in such a way that it is dependent on anything that is called only to check whether setup is to be performed.

Note: See Section 2.3.5 for further discussion of this rule.

Note: Function *jswtsr* may be used to call standard system rules.

Scheduler or Setup Time Rule

Description

The operation scheduler rule (run time) and setup time rule specified on an operation will be called to determine the run/setup time.

Synopsis

```
#include "factor.h"

double myrule(est, ldp, jsp, root)

int      est;     /* Estimate flag.            */
LOAD    *ldp;     /* Pointer to the load.      */
JOBSTEP *jsp;     /* Pointer to the operation. */
void    *root;    /* Expression root.          */
```

Returns

The run or setup time for the operation.

IInstall

The following function called from *ucini1* installs the rule as rule 39:

```
jsdfst (39, myrule);
```

Note: If the estimate flag is set, the load might not actually be at the operation that the run/setup time is being computed for. If the run/setup time rule relies on current conditions, it should return an appropriate estimate, since the conditions before the load actually does arrive at the operation might change.

Note: The input parameter *root* may be used to evaluate the run/setup time expression by calling function *evalexp*.

Note: Setup time rules are used to evaluate setup time for operations. Minimize setup selection rules often call this rule for setup duration computation with the estimate flag equal to 0 to get an accurate value.

Note: See Section 2.3.6 for further discussion of this rule.

Note: Function *jscmst* may be used to call standard system rules.

Batch Separation Rule

Description

The batch separation rule specified on the batch definition is called to determine which forming batch an arriving load should be added to.

Synopsis

```
#include "factor.h"

FORMBAT *myrule(ldp, bdp)

LOAD     *ldp;    /* Pointer to the arriving load.    */
BATCHDEF *bdp;    /* Pointer to the batch definition. */
```

Returns

Forming batch to add load to; NULL if no suitable forming batch found.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfbs (39, myrule);
```

Note: See Section 2.3.7 for further discussion of this rule.

Note: Function *seslfb* may be used to call standard system rules.

Batch Release Rule

Description

The batch release rule specified on the batch definition is called to determine what quantity should be added to the batch when the load is added to the batch.

Synopsis

```
#include "factor.h"
```

```
double myrule(ldp, bdp)

LOAD     *ldp;    /* Pointer to the arriving load.    */
BATCHDEF *bdp;    /* Pointer to the batch definition. */
```

Returns

Quantity to add to the batch.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfbq (39, myrule);
```

Note: See Section 2.3.8 for further discussion of this rule.

Note: Function *sequfb* may be used to call standard system rules.

Batch Override Rule

Description

The batch override release rule specified on the batch definition is called to determine if a forming batch should be released despite the fact that the release criteria have not been met.

Synopsis

```
#include "factor.h"

int myrule(fdp, bdp)

FORMBAT  *fdp;    /* Pointer to the forming batch.    */
BATCHDEF *bdp;    /* Pointer to the batch definition. */
```

Returns

Non-zero if the batch should be released; zero otherwise.

Install

The following function called from *ucini1* installs the rule as rule 39:

```
sedfov (39, myrule);
```

Note: If ADDOVFG on the batch definition is yes, the override release rule is called whenever a load is added to the batch.

Note: If PEROVFG on the batch definition is yes, the override release rule is called periodically. The period is defined by OVCYCLE on the batch definition. The override review event is scheduled to occur OVCYCLE units after the first forming batch is created at the batch definition. The event reschedules itself unless there are no more forming batches for the batch definition. Therefore, the override events do not occur during times when there is nothing to consider.

Note: See Section 2.3.9 for further discussion of this rule.

Note: Function *seadov* may be used to call standard system rules.

User-Writable Scheduling Rule Support Functions

This section presents the user-writable functions that support the actual rules that are written. Each function is applicable to a specific Scheduler event such as the start of the scheduling run, the creation of a new load, or saving output. By writing a function, you can embellish the processing that is performed when the associated scheduling event occurs. The scheduling functions which are invoked when given events occur during the scheduling, are presented in alphabetical order. These events are model load, model save, create and terminate a load, create a batch or regular in-process load, setup determination, and batch forming. For assistance in writing a function and installing a rule, see Section 2. The support functions are presented in the table below.

Function Name	Descriptioni
ucend	Called during exit from Scheduler
ucfin1	Called just before summaries are written in finalization
ucfin2	Called just after summaries are written
ucini1	Called after global variables are read
ucini2	Called after all input is read
ucjbtra	Called when an operation starts, finishes, gets interrupted or restarts after being interrupted
ucnwld	Called when a load is being initialized
ucnwor	Called when a job is being initialized
ucrlfb	Called when a forming batch load is released for processing
ucrstra	Called immediately before a resource changes to a new state

Function Name	Descriptioni
ucsini1	Called just before input is saved
ucsini2	Called just after input is saved
ucstib	Called when a new batch load is created while reading load status
ucstil	Called when a new load is created while reading load status
ucstring	Can be used in expressions throughout the scheduling model
uctmld	Called when a load is terminated
uctmor	Called when a job is terminated
ucvalue	Can be used in expressions throughout the scheduling model
ucwtsr	Called from setup time rules to determine whether setup is necessary

ucend

Description

Function *ucend* is called only once upon exit from the Scheduler

Synopsis

```
#include "factor.h"

void ucend(void)
```

Returns

Nothing.

Note: This function is typically used to perform cleanup activities, such as closing user files.

ucfin1

Description

Function *ucfin1* is called at the end of a Scheduler before summaries are written.

Synopsis

```
#include "factor.h"

void ucfin1(void)
```

Returns

Nothing.

Note: This function is typically used to perform final computations and write out custom information at the end of a Scheduler.

Note: In most cases, user finalization code could be placed in either function *ucfin1* or function *ucfin2* with the same effect.

Note: This function is called at the end of each scheduler run (unless the run was aborted). It is also called before saving output in the Scheduler.

ucfin2

Description

Function *ucfin2* is called at the end of a scheduler run (unless it was aborted) after summaries are written, and after saving output in the Scheduler.

Synopsis

```
#include "factor.h"

void ucfin2(void)
```

Returns

Nothing.

Note: This function is typically used to perform final computations and write out custom information at the end of a Scheduler. In most cases, user finalization code could be placed in either function *ucfin1* or function *ucfin2* with the same effect.

ucini1

Description

Function *ucini1* is called at the beginning of a scheduling run before the alternative information has been read.

Synopsis

```
#include "factor.h"

void ucini1(void)
```

Returns

Nothing.

Note: This function is typically used to install custom rules and similar initialization tasks that do not depend upon the model data. It is recommended that you only call rule “install” functions from this routine.

Note: This function is called once in the Scheduler before model data is loaded.

ucini2

Description

Function *ucini2* is called at the beginning of a scheduling run after all model information has been read.

Synopsis

```
#include "factor.h"

void ucini2(void)
```

Returns

Nothing.

Note: This function is typically used to perform initialization actions that depend on model data.

Note: This function is called at the beginning of each execution of the Scheduler.

ucjbtra

Description

Function *ucjbtra* is called when an operation starts, finishes, gets interrupted or restarts after being interrupted.

Synopsis

```
#include "factor.h"

void ucjbtra(pstLoad, cJobState)

LOAD *pstLoad;   /* Pointer to the current load. */
char  cJobState; /* Character code for the job state, S, I, R, E, A. */
```

Returns

Nothing.

Note: This function can be used to trace a load through the system.

Note: The parameter *cJobState* can have the following values:

- S - the operation processing is starting.
- I - the operation is being interrupted.
- R - the operation is being restarted after being interrupted.
- E - the operation is at the end.
- A - arrival, or re-arrival, to operation processing functions (this will occur multiple times during the processing of a single operation).

Note: You will get duplicate calls to *ucjbtra* with different codes. Each time *ucjbtra* is called with codes ‘R’ or ‘I’ there will also be a call with ‘A’. When called with ‘S’ you may also have a call with ‘A’. You may also get a call with the ‘A’ code by itself. The function will get called only once with the ‘E’ code. This routine can be called from *ucfin1* to write out partially completed loads as well.

ucnwld

Description

Function *ucnwld* is called whenever the Scheduler creates a new load.

Synopsis

```
#include "factor.h"

void ucnwld(ldp)

LOAD *ldp;        /* Pointer to the load just created. */
```

Returns

Nothing.

Note: New loads are created when jobs are released and at the following operations: BATCH and ACCUMULATE/SPLIT (These types are created under the covers).

Note: This function is typically used to attach auxiliary data to the new load.

ucnwor

Description

Function *ucnwor* is called whenever the Scheduler creates a new job.

Synopsis

```
#include "factor.h"

void ucnwor(orderp)

ORDER *orderp;        /* Pointer to the job just created. */
```

Returns

Nothing.

Note: This function is typically used to attach auxiliary data to the new job.

ucrlfb

Description

Function *ucrlfb* is called when a forming batch is released.

Synopsis

```
#include "factor.h"

void ucrlfb(formbat, batdef)

FORMBAT  *formbat;       /* Pointer to the forming batch.    */
BATCHDEF *batdef;        /* Pointer to the batch definition. */
```

Returns

Nothing.

Note: This function is typically used to attach auxiliary data to a batch load just before the batch load is released.

ucrstra

Description

Function *ucrstra* is called immediately before a resource changes to a new state.

Synopsis

```
#include "factor.h"

void ucrstra(pstRes, pstLoad, newstate)

RESRC *pstRes;    /* Pointer to the resource that changed. */
LOAD  *pstLoad;   /* Pointer to the current load. */
```

```
int    newstate;  /* New state. */
```

Returns

Nothing.

Note: This function can be used to trace the state of a resource during a Scheduler. The new state value is a constant defined in *factor.h* and can be changed by calling the function *secsrs (see Section 6.11)*. See the table below for the states:

State	Description
RS_PROC	Processing.
RS_SETUP	Undergoing setup.
RS_BLOCKED	Allocated but not processing.
RS_IDLE	Idle.
RS_BREAK	Broken down.
RS_MAINT	Undergoing maintenance.
RS_OFFSHIFT	Off shift.

The old state can be obtained from the resource structure, *pstRes->rsstate*.

It is possible that *pstLoad* will be NULL when this function is called.

ucsini1

Description

Function *ucsini1* is called at the beginning of save in the Scheduler.

Synopsis

```
#include "factor.h"

void ucsini1(void)
```

Returns

Nothing.

Note: This function is typically used to prepare user auxiliary data prior to save.

Note: In most cases, user code could be placed in either function *ucsini1* or function *ucsini2* with the same effect.

ucsini2

Description

Function *ucsini2* is called after save in the Scheduler.

Synopsis

```
#include "factor.h"

void ucsini2(void)
```

Returns

Nothing.

Note: This function is typically used to write out user input auxiliary data tables at the end of saving a model.

Note: In most cases, user save code could be placed in either function *ucsini1* or function *ucsini2* with the same effect.

ucstib

Description

Function *ucstib* is called when an in-process batch load is created.

Synopsis

```
#include "factor.h"

void ucstib(bldp)
```

```
LOAD *bldp;       /* Pointer to the newly created load. */
```

Returns

Nothing.

Note: This function is typically used to attach auxiliary data to an in-process batch load.

ucstil

Description

Function *ucstil* is called when an in-process load is created.

Synopsis

```
#include "factor.h"

void ucstil(ldp)

LOAD *ldp;        /* Pointer to the newly created load. */
```

Returns

Nothing.

Note: This function is typically used to attach auxiliary data to an in-process load.

ucstring

Description

Function *ucstring* can be used in expressions throughout the scheduling model.

Synopsis

```
#include "factor.h"
```

```
void ucstring(icase, argvalue, type, szBuf, iMax)

int    icase;     /* Integer parameter to function. */
void * argvalue;  /* Pointer to the second argument. */
char   type;      /* Type of the second argument. */
char  *szBuf;     /* Output: string to return. */
int    iMax;      /* Maximum size of the string to be returned. */
```

Returns

Nothing.

Note: This function can be used to return strings and names for use in lookup tables or comparison expressions. A reference to *ucstring* is placed in a system expression. This reference has 2 parameters: 1) *icase*, an integer (corresponds with *icase* defined above) and 2)*express*, an expression. The user then writes a *ucstring* function with 5 parameters: 1) *icase*, upon which the function will branch (switch, if-then-else, etc.) if there are multiple different *ucstring* references in expressions, 2) *argvalue*, result of an evaluation of the expression (*express*); 3) *type*, the type of the expression ('R' = double *, 'I' = int*, 'S' = char*), 4) *szBuf*, the string to return from the function (output), and 5) *iMax*, the maximum size of the string to be returned (the default is 255 characters).

Note: The value *argvalue* will never be "NULL" or (void*) NULL.

uctmld

Description

Function *uctmld* is called when a load is being terminated.

Synopsis

```
#include "factor.h"

void uctmld(ldp)

LOAD *ldp;        /* Pointer to the load being terminated. */
```

Returns

Nothing.

Note: This function is typically used to free data attached to the auxiliary data pointer on a load.

uctmor

Description

Function *uctmor* is called when an order is being terminated.

Synopsis

```
#include "factor.h"

void uctmor(orderp)

ORDER *orderp;        /* Pointer to the order being terminated. */
```

Returns

Nothing.

Note: This function is typically used to free data attached to the auxiliary data pointer on an order.

ucvalue

Description

Function *ucvalue* can be used in expressions throughout the scheduling model.

Synopsis

```
#include "factor.h"

double ucvalue(icase, argvalue, type)

int    icase;     /* Integer parameter to function. */
void * argvalue;  /* Pointer to the second argument. */
char   type;      /* Type of the second argument. */
```

Returns

Value to use in expression.

Note: This function can be used to compute simple values or do more complex functions. A reference to *ucvalue* is placed in a system expression. This reference has 2 parameters: 1) *iCase*, an integer (corresponds with *icase* defined above) and 2) *express*, an expression. The user then writes a *ucvalue* function with 3 parameters: 1) *icase*, upon which the function will branch (switch, if-then-else, etc.) if there are multiple different *ucvalue* references in expressions, 2) *argvalue*, the result of an evaluation of the expression (*express*), and 3) *type*, the type of the expression ('R' = double*, 'I' = int*, 'S' = char*).

Note: The value argvalue will never be "NULL" or (void *)NULL.

ucwtsr

Description

Function *ucwtsr* is called from setup time rules to determine whether setup is necessary.

Synopsis

```
#include "factor.h"

int ucwtsr(ldp, type, rp, jsp, rule)

LOAD    *ldp;     /* Pointer to the load.             */
char     type;    /* 'R'esource                       */
void    *rp;      /* Pointer to the resource.         */
JOBSTEP *jsp;     /* Pointer to the operation.        */
int      rule;    /* When-to-setup rule to use.       */
```

Returns

Non-zero if setup is required; zero otherwise

Chapter 5: Global Variables

5

Global variables are rarely used in user code, but several are provided for use under certain circumstances. The cs* variables are used when handling the event calendar, and starting and stopping the Scheduler. The ssgvar variable is the root of all system scheduler data structures. These variables are presented in alphabetical order. For each variable, there is a brief description, the type of variable, and notes about its use.

Variable	Description	Type	Notes
cscln0	Variable *cscln0* is the list of internal scheduler events, often called the internal event calendar.	CSLIST *cscln0;	This list should be accessed using the special functions intended for that purpose, rather than standard list functions, whenever possible.
csclnr	Variable csclnr is the list of regular scheduler events, often called the regular event calendar.	CSLIST *csclnr;	This list should be accessed using the special functions intended for that purpose, rather than standard list functions, whenever possible.
cshalt	Variable *cshalt* is the scheduler halt flag. Normally this flag is set to false (zero). If it is set to a true (non-zero) value during the processing of an event, the scheduler will terminate immediately after that event returns.	int cshalt;	This variable is not normally used from user-written code but may be useful if the end of the scheduler is dictated by events other than the end of the scheduling window or the completion of the last order.

Variable	Description	Type	Notes
cstbeg	Variable *cstbeg* is the beginning time of the scheduler, in hours, from time zero.	double cstbeg;	Currently, the scheduler always starts at time zero. Therefore, this value is always zero.
cstfin	Variable *cstfin* is the ending time of the scheduler, in hours, from the start of the scheduler.	double cstfin;	Currently this value is the difference, in hours, between the start and end of the scheduler or, in other words, the length of the scheduler in hours.
cstnow	Variable *cstnow* is the current scheduler time, in hours, from the start of the scheduler.	double cstnow;	This is the elapsed time, in hours, since the start of the scheduler.
ssgvar	Variable *ssgvar* is the main system global data structure. It contains, through substructures and lists, the entire system model. This data structure is quite complex; therefore, see the include file “factor.h” for a complete description of its members.	SSGLOBL ssgvar;	See note below.

Note: Major components of *ssgvar*:
sgctrl - Global scheduler control data structure (i.e., scheduler start times, trace information, etc.) and system output data collection flags and dataset numbers.
sgflg - Structure for system specific output data collection flags and dataset numbers.
global lists - Lists of all components in the scheduler model.
hash tables - Hash tables for data input.
misc. controls - Various global values used by the scheduler. For example, sgestfg is the estimate processing time flag used by random variable functions to signal them to return the mean of the distribution, if estimating.

Chapter 6: User Callable Functions

6

This section presents the user-callable support functions. You can use these functions to embellish the Scheduler. The user-callable functions are presented by category. You can get the parameters of each function from the *fproto.h* file. The functions are categorized by:

- Batch Functions.
- Date and Time Functions.
- Entity Management and Event Scheduling Functions.
- Find Functions.
- Install Functions.
- List Manipulation Functions.
- Load Functions.
- Miscellaneous Functions.
- Operation Event Functions.
- Operation Support Functions.
- Resource Functions.
- Resource Group Functions.
- System Status Functions.

Some functions may be listed more than once if they fall into more than one category.

Batch Functions

Function	Description
seadov	Cause an override release review of a forming batch load.
sedfbq	Install a batch release function.
sedfbs	Install a batch separation function.
sedfov	Install a batch override function.
sefdbt	Find a batch definition.
sefdfb	Find a forming batch load.
semkfb	Make a forming batch load.
semvbt	Move the possessions of a member load to its parent batch load.
sequfb	Compute the forming batch quantity of a load.
serlfb	Release a forming batch load.
Seslfb	Select a forming batch load to accumulate a given load in

Date and Time Functions

Function	Description
gedi2j	Convert date and time integers into a Julian representation.
gej2di	Convert a Julian representation into date and time integers.
gej2ds	Convert a Julian representation into a “MM-DD-YY HH:MM” form.
gest2j	Returns the current system time as a Julian representation.
gewkdy	Return the day of the week given a Julian representation.
ses2di	Convert a time in hours from start into date and time integers.
ses2ds	Convert a Scheduler time into a “MM-DD-YY HH:MM” form.
ses2st	Convert a Scheduler time into a “MM-DD-YY HH:MM:SS” form.

Entity Management and Event Scheduling Functions

Function	Description
cschd0	Schedule a system event on the internal event calendar.
csched	Schedule a system event on the regular event calendar.
cselep	Search the event lists for an entity.
cselfv	Search the event lists for the first entity scheduled for a given function.
cselnv	Search the event lists for the next entity scheduled for a given function.
csepea	Get the scheduled event address for an entity.
csepet	Get the scheduled event time for an entity.
csnew	Get a pointer to a new entity of the specified size.
CSNEW	Get a pointer to a new entity of the specified size.
csterm	Terminate an entity pointer.
uccschd0	Schedule a user event on the internal event calendar.
uccsched	Schedule a user event on the regular event calendar.

Find Functions

Function	Description
sefdat	Find a load attribute.
sefdbt	Find a batch definition.
sefdfb	Find a forming batch load.
sefdil	Find an in-process load.
sefdjs	Find an operation.
sefdlk	Find a lookup table (setup matrix).
sefdor	Find a job.
sefdpr	Find a routing.
sefdpt	Find an item.
sefdrg	Find a resource group.
sefdrq	Find a request matching a given load in a given list.
sefdrs	Find a resource.
sefdrt	Find a resource tag matching a given resource in a given list.
sefdry	Find a resource type.
sefdsh	Find a shift schedule.

Install Rule Functions

Function	Description
expdfr	Install an expression evaluation function.
jsdfst	Install an operation (run or setup) time function.
sedfbq	Install a batch release function.
sedfbs	Install a batch separation function.
sedfgs	Install a resource group allocation function.
sedfok	Install an job ranking function.
sedfov	Install a batch override function.
sedfrk	Install a resource sequence function.
sedfsl	Install a resource selection function.

List Manipulation Functions

Function	Description
csdlls	Delete a list.
csdsst	Disable statistics collection for a list.
csenst	Enable statistics collection for a list.
csfsls	Get a pointer to the first entity in a list.
csgpls	Remove an entity from a list by pointer.
csgtls	Remove an entity from a list by number.
csinls	Determine whether an entity is in a list.
cslsls	Get a pointer to the last entity in a list.
csmkls	Create and initialize a list with statistics.
CSMXLS	Create and initialize a list without statistics.
csnxls	Get a pointer to the next entity in a list.
cspfls	Place an entity into a list after another entity.
cspols	Place an entity into a list using a specified ordering.
csppls	Place an entity into a list before another entity.
csprls	Get a pointer to the previous entity in a list.
csptls	Place an entity into a list using the list’s ordering.
cssols	Sort a list.
csssls	Get a pointer to the time-persistent statistics for a list.
csszls	Get the current size of a list.
cswsls	Get a pointer to the observed statistics for a list.

Load Functions

Function	Description
seajld	Adjust the accumulators and resource counts for a load.
seacsq	Turn on the sequencing rule for a request list.
secmds	Compute dynamic slack for a load.
secrcl	Create resource capacity list for a load.
secrwcl	Create WIP capacity list for a load.
sedfrk	Install a load ranking function.
sefdil	Find an in-process load.
sefdrq	Find a request matching a given load in a given list
selork	Return the ranking of a load.
semcmp	Check a load for exceeding the maximum overrun for a resource.
semvld	Move some or all of the possessions of a load to another load.
senwld	Create a load.
sermot	Compute the number of remaining operations and run time for a load.
setmld	Terminate a load.
setosq	Turn off the sequencing rule for a request list.
batchid	Batch ID if batch load.
batchnam	Batch name if member of batch
jobarriv	Specific load arrival time at operation.
jobstart	Specific load start time at operation.
jsname	Specific load current operation.
ldduedate	Specific load due date

ldinsys	Number of loads in the system.
ldlngjstep	Specific load’s longest remaining operation
ldoptime	Specific load’s operation time at operation
ldordnmld	Specific load’s order’s number of load
ldordsize	Specific load’s order size
ldprior	Specific load priority
ldprtime	Specific load’s run time at operation
ldqutime	Specific load’s wait time at operation
ldreldate	Specific load’s job’s release date
ldresalloc	Specific load has resource allocated (1=yes, 0=no)
ldrmjsteps	Specific load’s remaining number of operations
ldrmprtime	Specific load’s remaining processing time
ldsttime	Specific load’s setup time at this operation
lkupname	Specific load’s item lookup table name.
loaddone	Number of loads completed.
loadid	Specific load ID.
loadqtim	Specific load total queue time.
loadproc	Number of loads processing.
loadsize	Specific load size.
loadwait	Number of loads waiting.
ordernam	Specific load job name.
partfam	Specific load part family name.
partname	Specific load part name.
partsfam	Specific load part subfamily name.
proctime	Specific load processing time.

remproc	Specific load remaining processing time.
rgindex	Specific load index of member from resource group allocated to load.
rgmem	Specific load name of resource from group allocated to load.

Miscellaneous Functions

Function	Description
evalexp	Evaluate an expression.
seepor	End processing for a job.
s*eferr*	Report a fatal error message.
seissd	Return whether shift is down.
serlld	Release loads for a job.
setrace	Report a debug trace message.
setrev	Event function to change the trace level.
setrlv	Change the trace level.
sewarn	Report non-fatal warning.
sierr	Report fatal input error.
siorrk	Return the ranking of a job.
sistsv	Allocate storage for and copy a string.
siwarn	Report non-fatal input warning.
sticslist	Retrieve a CSLIST from a status file.
stipointer	Retrieve a buffer from a status file.
stocslist	Save a CSLIST to a status file.
stopointer	Save a buffer to a status file.
getxcell	Get a value from an Excel spreadsheet.
lookup	Get a value from a lookup table (setup matrix).
orddone	Number of jobs completed.
prtdone	Number of parts completed.
prtinsys	Number of parts in the system.
prtproc	Number of parts processing in the system.

prtwai	Number of parts waiting in the system.
simend	Ending Scheduler time (hours from start).
simnow	Current Scheduler time (hours from start).
statsclr	Time statistics were last cleared (hours from start).

Operation Event Functions

Function	Description
seinjs	General interrupt of a operation event function.
sejsev	Schedules a load to arrive at a operation.

Operation Support Functions

Function	Description
jsaloc	Allocate resources for an operation.
jsalrs	Allocate the *n*th resource/resource group on an operation.
jsavrs	Get the availability of the *n*th resource/resource group on an operation.
jsclst	Update load statistics during operation processing.
jscmjt	Return the run time of a load for an operation.
jscmst	Return the operation portion of step time of load for an operation.
jscmsu	Return the setup part of step time of a load for an operation.
jscqrs	Cancel requests for the *n*th resource/resource group on an operation.
jsdfst	Install run/setup time function.
jsfree	Handle all of the operation resource free phases.
jsfrhn	Free the handle returned by function *jsavrs*.
jsfrrs	Free the *n*th resource/resource group on a operation.
jsfsbt	Select the first operation on batch routing.
jsfsjs	Select the first operation on routing.
jsinrs	Interrupt the use of a resource by a load.
jsnxbt	Select next operation after a batch routing.
jsnxjs	Select next operation in routing.
jsrars	Retrieve resources preempted due to off shift or failure.

jsrqrs	Request the *n*th resource/resource group on a operation.
jsrsen	Reschedule end of operation after interruption.
jsscen	Schedule the end of service for a operation.
jswtsr	Determine whether setup required.
sefdjs	Find operation.
sejssl	Determine whether to process this operation.

Resource Functions

Function	Description
seaars	Returns resource availability status to a load. (Considers must-complete and maximum-overrun.)
sealrs	Allocate a resource to a load.
seavrs	Returns resource availability status. (Does not consider must-complete and maximum-overrun.)
seckrs	Checks the queue for a resource.
secqrs	Cancel a request for a resource to a load.
secsrs	Change resource state. (See states in Section 4.2, function *ucrstra*)
sedfsl	Install a resource request selection function.
sefdrs	Find a resource.
sefdrt	Find a resource tag matching a given resource in a given list.
sefdry	Find a resource type.
sefrrs	Free a resource from a load.
seissd	Determines whether a resource is off shift.
seprct	Returns the projected completion time of a job.
serqor	Returns relationship of resource requests.
serqrs	Request a resource for a load.
serqsl	Reorders the resource request list before a request is selected.
seutrs	Return resource utilization.
sexfrl	Transfer a list of resources to a load.
sexfrs	Transfer a specific resource to a load.

siadhr	Add a held resource to an in-process load.
siadrf	Add a resource reference to a operation.
siarsh	Add a shift reference to a resource.
raveqlen	Average queue length for a resource.
raveqtim	Average waiting time for a resource.
rcurqlen	Current queue length for a resource.
resstat	Current resource status.
rscrblck	Current resource blocked time proportion.
rscrbusy	Current resource busy time proportion.
rscreset	Current resource setup time proportion.
rscridle	Current resource idle time proportion.
rscroff	Current resource off-shift time proportion.
rscurshift	Resource current shift.
rsonblck	Current resource on-shift blocked time proportion.
rsonbusy	Current resource on-shift busy time proportion.
rsonidle	Current resource on-shift idle time proportion.
rsonset	Current resource on-shift setup time proportion.
rsprct	Estimated time to complete current operation.
rstmeos	Time to end of “up” period.

Resource Group Functions

Function	Description
seadsl	Add a resource to selection list.
sealrg	Allocate a resource from a resource group to a load.
secqrg	Cancel a request for a resource group to a load.
sedfgs	Install a resource group member selection function.
sefdrg	Find a resource group.
sefmrg	Free list of members to allocate.
sefrrg	Free a number of units of a resource group for a load.
sermsl	Remove a resource from selection list.
serqrg	Request a number of units of a resource group for a load.
sergms	Select a member from a resource group for a load (called from a member selection rule).
seslrg	Select a member from a resource group for a load (called from an allocation rule).
siadrf	Add a resource group reference to a operation.
rgavqlen	Average queue length for a resource group.
rgavqtim	Average waiting time for a resource group.
rgblock	Current number of blocked resources in a resource group.
rgbusy	Current number of busy resources in a resource group.
rgcrblck	Current resource group blocked time proportion.
rgcrbusy	Current resource group busy time proportion.
rgcridle	Current resource group idle time proportion.

rgcroff	Current resource group off-shift time proportion.
rgcrset	Current resource group setup time proportion.
rgcurqln	Current queue length for a resource group.
rgidle	Current number of idle resources in a resource group.
rgoff	Current number of off-shift resources in a resource group.
rgonblck	Current resource group on-shift blocked time proportion.
rgonbusy	Current resource group on-shift busy time proportion.
rgonidle	Current resource group on-shift idle time proportion.
rgonset	Current resource group on-shift setup time proportion.
rgsetup	Current number of setup resources in a resource group.

System Status Functions

Function	Description
batchid	Batch ID if batch load.
batchname	Batch name if member of batch.
getxcell	Get a value from an Excel spreadsheet.
jobarriv	Specific load arrival time at operation.
jobstart	Specific load start time at operation.
jsname	Specific load current operation.
ldinsys	Number of loads in the system.
lddrtime	Load’s drop off time at this operation.
ldduedate	Load’s order’s due date.
ldlngjstep	Load’s longest remaining operation.
ldoptime	Load’s operating time at this operation.
ldordnmld	Load’s order’s number of loads.
ldordsize	Load’s order size.
ldpktime	Specific loads pickup time at operation
ldprior	Specific load priority
ldprtime	Specific load’s processing time at operation
ldqutime	Specific load’s wait time at operation
ldreldate	Specific load’s order’s release date
ldresalloc	Specific load has resource allocated (1=yes, 0=no)
ldrmjsteps	Specific load’s remaining number of operations
ldrmprtime	Specific load’s remaining processing time
ldsttime	Specific load’s setup time at this operation
lkupname	Specific load’s part lookup table name.

loaddone	Number of load completed.
loadid	Specific load ID.
loadqtim	Specific load total queue time.
loadproc	Number of loads processing.
loadsize	Specific load size.
loadwait	Number of loads waiting.
lookup	Get a value from a lookup table.
mcrprct	Estimated time to complete operation
mctmeos	Time to end of up period.
orddone	Number of orders completed.
ordernam	Specific load order name.
partfam	Specific load part family name.
partname	Specific load part name.
partsfam	Specific load part subfamily name.
prtdone	Number of parts completed.
prtinsys	Number of parts in the system.
prtproc	Number of parts processing in the system.
prtwait	Number of parts waiting in the system.
proctime	Current load processing time.
raveqlen	Average queue length for a resource.
raveqtim	Average waiting time for a resource.
rcurqlen	Current queue length for a resource.
remproc	Current load remaining processing time.
resstat	Current resource status.
rgavqlen	Average queue length for a resource group.
rgavqtim	Average waiting time for a resource group.

rgblock	Current number of blocked resources in a resource group.
rgbusy	Current number of busy resources in a resource group.
rgcrblck	Current resource group blocked time proportion.
rgcrbusy	Current resource group busy time proportion.
rgcridle	Current resource group idle time proportion.
rgcroff	Current resource group off-shift time proportion.
rgcrset	Current resource group setup time proportion.
rgcurqln	Current queue length for a resource group.
rgmem	Specific load name of resource from group allocated to load.
rgidle	Current number of idle resources in a resource group.
rgoff	Current number of off-shift resources in a resource group.
rgonblck	Current resource group on-shift blocked time proportion.
rgonbusy	Current resource group on-shift busy time proportion.
rgonidle	Current resource group on-shift idle time proportion.
rgonset	Current resource group on-shift setup time proportion.
rgsetup	Current number of setup resources in a resource group.
rscrblck	Current resource blocked time proportion.
rscrbusy	Current resource busy time proportion.
rscridle	Current resource idle time proportion.
rscroff	Current resource off-shift time proportion.

rscrset	Current resource setup time proportion.
rscurshift	Resource current shift.
rsonblck	Current resource on-shift blocked time proportion.
rsonbusy	Current resource on-shift busy time proportion.
rsonidle	Current resource on-shift idle time proportion.
rsonset	Current resource on-shift setup time proportion.
rsprct	Estimated time to complete current operation.
rstmeos	Time to end of “up” period.
simend	Ending scheduler time (hours from start).
simnow	Current scheduler time (hours from start).
statsclr	Time statistics were last cleared (hours from start).

Chapter 7: Making User-Defined Rules Available to the Scheduler

7

This section describes how to create the dll file that the Scheduler references to obtain the custom user code. Note that user code compiled for the Scheduler applies to all alternatives in the database. The install procedure creates a folder named USERCODE that contains the user code support files. You should create a subfolder beneath this one with the same name as your SQL database to hold your user code source files. The Scheduler also looks for the user code DLL file (USER.DLL) in this directory. USER CODE = Rules + Supporting code

Writing User Code for Unicode

The Scheduler uses Unicode to support international string issues. To write code that can be conditionally compiled for Unicode, MBCS, or neither, follow these programming guidelines:

- Use the _**T** macro to code literal strings conditionally to be portable to Unicode. For example:
- psqlda = dboptab **(_T("MYBOM")**, p_ssgvar->sgctrl.scprtds, DB_FETCH)
- When you pass strings, pay attention to whether function arguments require a length in characters or a length in bytes. The difference is important if you're using Unicode strings.
- Use portable versions of the C run-time string-handling functions. See the section on String Manipulation in the Microsoft Visual C/C++ documentation for a complete list and for further information.

See the table below for examples.

Use This	Instead of
_tcscpy	strcpy
_tcsncpy	strncpy
_tcscmp	strcmp
_tcsncmp	strncmp
_tcscat	strcat

Use This	Instead of
_tcsncat	strncat
_tcschr	strchr
_stprintf	sprintf

Use the following data types for characters and character pointers:

- TCHAR where you would use char.
- LPTSTR or TCHAR * where you would use char*.
- LPCTSTR where you would use const char*.

Compiling and Linking Scheduler User Code

To compile and link user code for the Scheduler, Microsoft Visual C++ Version 6.0 SP3 or later (version 8.02 or earlier) or Microsoft Visual Studio 2010 (version 8.03) is required. If you did not install Microsoft Visual C++ (version 8.02 or earlier) or Microsoft Visual Studio (version 8.03) with the option to set up for use from a command prompt, your PATH, LIB, and INCLUDE environment variables may not be set correctly. To set these, you can run VCVARS32.BAT, which is located in the \bin subdirectory of your Visual C++ (version 8.02 or earlier) or Visual Studio (version 8.03) installation.

The following steps will guide you in the procedures used to compile and link your user code:

1 Create your database directory and copy the files MAKEFILE and USER.DEF from the USERCODE directory into it. For example:

- $ cd USERCODE
- $ mkdir mydb
- $ cd mydb
- $ copy ..\MAKEFILE .
- $ copy ..\USER.DEF .

2 Edit the MAKEFILE and change the “OBJFILES” line, which is near the top of the file, to list your .C user code files. Instead of a .C file extension, you must use a .OBJ file extension when specifying the files. For example, the files FILE1.C, FILE2.C, and FILE3.C would be specified as:

```
OBJFILES = file1.obj file2.obj file3.obj
```

Case does not matter. In most cases, you will not need to change anything else in this file. However, there are additional variables you can use for custom compile or link options or for additional libraries to be linked into the program.

3 If your user code calls functions uccschd0 or uccsched you must perfor m this step. Otherwise, you can skip to step 4. When scheduling user events, one of the steps is to add function declaration lines to the .DEF file that is used by the linker. See the documentation on function uccsched in Section 3.3 for more details. Instead of copying and editing the APS_SUSR.DEF file

as documented there, you must make your edits to the USER.DEF file that you copied in step 1. Go to the end of the USER.DEF file and add a line consisting of the name of the user-written event function. The case should match the case of the name as it appears in the .C file. Repeat for each user-written event function.

4 You can now perform the compile and link by running the NMAKE command. At an MSDOS command prompt, change directory to the database directory and run the NMAKE command. This command reads the file MAKEFILE and follows the directives in it to compile and link your user code. Be sure to correct any compile or link errors before proceeding. For example:

```
$ cd USERCODE\mydb
$ nmake
```

5 You can now run the Scheduler and it will use your user code. Any time you change your user code, you must repeat the appropriate steps in this process.

Debugging Scheduler User Code (version 8.02 or earlier)

1 Create the project:

   a Start the Microsoft Visual C++ 6.0 Developer's Studio. Select File/New, and then select the Projects tab.

   b Select Win32 Dynamic-Link Library and give the new project a name.

   c Choose to create a new workspace and click OK.

   d Choose to create an empty DLL project and then click on Finish.

2 Add files to project:

   a Select Project/Add to Project/Files.

   b Add your .C files.

   c Add the USER.DEF file from the same directory.

   d Add the following .LIB files from the USERCODE directory:

   - aps_sim.lib
   - aps_susr.lib
   - aps_ulib.lib
   - aps_util.lib

3 Modify the project settings:

   a Select Project/Settings and then the C/C++ tab.

   - For the Category, select Preprocessor.
   - For the Additional include directories edit box, type in the full path to the USERCODE directory.

- Add the following to the Preprocessor definitions:
- UNICODE,_UNICODE,_WIN32,SYTEAPS,_AFXDLL,_DLL
- Remove _MBCS from the Preprocessor definitions.
- For the Category, select Code Generation.
- For the Use run-time library, select Multithreaded DLL.

b Select the Link tab.

- For the Category, select General.
- For the Output file name edit box, type in the full path and USER.DLL. For example, if your database is called MYDB, you would type: *C:\Program Files\Infor\APS\Scheduler\USERCODE\MYDB\USER.DLL*

c Select the Debug tab.

- For the Category, select General.
- For the Executable for debug session, enter the full path to APS_BATS.EXE. For example: *C:\Program Files\Infor\APS\Scheduler\APS_BATS.EXE* .
- For the Working directory, specify the directory where APS_BATS.EXE is located. For example: *C:\Program Files\Infor\APS\Scheduler* .
- For the Program Arguments, enter a line that looks like the following (substitute your User Id, Password, Server name, Database name, and the correct alternative number): The only space should be in front of the alternative number. For example: *UID=me;PWD=mypassword;SERVER=myserve;DATABASE=mydb; 0 .*

d Select OK to accept changes.

4 Run the project:

a Select Build/Build user.dll. You should not see any errors.

b Click on the FileView tab and expand the Source Files item.

c Double-click on the file name you want to debug.

d Click on the line where you want the code to stop and then click on the hand icon to Insert/ Remove Breakpoint(F9). This will set a breakpoint on this line.

e Select Build/Start Debug and Go. The Scheduler will start running and pause when it reaches your breakpoint.

Debugging Scheduler User Code (version 8.03)

1 Create the project:

a Start Microsoft Visual Studio. Select File/New Project.

b Select Win32 Project and give the new project a name.

c Choose to create a new solution and click OK.

d Choose to create an empty DLL project and then click on Finish.

2 Add files to project:

   a Select Project/Add Existing Item.

   b Add your .C files.

   c Add the USER.DEF file from the same directory.

   d Add the following .LIB files from the USERCODE directory:

   - aps_sim.lib
   - aps_susr.lib
   - aps_ulib.lib
   - aps_util.lib

3 Modify the project settings:

   a Select Project/Properties and then the C/C++ tab.

   - For the Category, select Preprocessor.
   - For the Additional include directories edit box, type in the full path to the USERCODE directory.
   - Add the following to the Preprocessor definitions: *UNICODE,_UNICODE,_WIN32,SYTEAPS,_AFXDLL,_DLL*.

   b Select the Link tab.

   - For the Category, select General.
   - For the Output file name edit box, type in the full path and USER.DLL. For example, if your database is called MYDB, you would type: *C:\Program Files\Infor\APS\Scheduler\USERCODE\MYDB\USER.DLL* .

   c Select the Debugging tab.

   - For the Category, select General.
   - For the Executable for debug session, enter the full path to APS_BATS.EXE. For example: *C:\Program Files\Infor\APS\Scheduler\APS_BATS.EXE* .
   - For the Working directory, specify the directory where APS_BATS.EXE is located. For example: *C:\Program Files\Infor\APS\Scheduler* .
   - For the Program Arguments, enter a line that looks like the following (substitute your User Id, Password, Server name, Database name, and the correct alternative number): The only space should be in front of the alternative number. For example: *UID=me;PWD=mypassword;SERVER=myserve;DATABASE=mydb; 0* .

   d Select OK to accept changes.

4 Run the project:

   a Select Build/Build <project name>. You should not see any errors.

   b Click on the FileView tab and expand the Source Files item.

   c Double-click on the file name you want to debug.

   d Click on the line where you want the code to stop and then click on the hand icon to Insert/Remove Breakpoint(F9). This will set a breakpoint on this line.

e Select Debug/Start Debugging. The Scheduler will start running and pause when it reaches your breakpoint.