WL-TN-90-04

WL-TN-90-04

(2)

SPHC MANUAL

AD-A220 743

R. Stellingwerf

Mission Research Corporation
Albuquerque, NM 87106

March 1990

Final Report

Weapons Laboratory
Air Force Systems Command
Kirtland Air Force Base, NM 87117-6008

WL-TN-90-04

This final report was prepared by Mission Research Corporation, Albuquerque, N.Mex., under Contract F29601-86-C-0216, Job Order 57972406 with the Weapons Laboratory, Kirtland Air Force Base, New Mexico. David W. Price (AWX) was the Laboratory Project Officer-in-Charge.

This report has been reviewed and is approved for publication.

David W. Price

DAVID W. PRICE
Project Officer

FOR THE COMMANDER

[illegible]

J. DOUGLAS BEASON
Maj, USAF
Chief, High Energy Plasma Div

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1990	Final Report - Mar 88-Dec 89

4. TITLE AND SUBTITLE

SPHC MANUAL

5. FUNDING NUMBERS

JON: 5797
TA: 24
WU: 06
PEN: 62601F
C: F29601-86-C-0216

6. AUTHOR(S)

Stellingwerf, R.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Mission Research Corporation
Albuquerque, NM 87106

8. PERFORMING ORGANIZATION REPORT NUMBER

MRC/ABQ-R-1237

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Weapons Laboratory
Kirtland AFB, NM 87117-6008

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

WL-TN-90-04

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Statement A

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document is a manual for the use of the Smooth Particle Hydrodynamics Code (SPHC). This code offers rapid hydrodynamic simulations with no large degradation in simulation accuracy. It has been successfully used to simulate implosions, explosions, shock tubes and to monitor aggregate masses under hydrodynamic conditions. In addition, SPHC can divide, create, and subtract particles in the simulations to optimize results without adversely affecting run conditions. The code is written in the C language and is relatively machine independent. It can be hosted on a variety of computers, ranging from supercomputers to IBM-AT class microcomputers. The SPHC manual has four sections: (1) a User's Guide; (2) a Programmar's Guide; (3) a Technical Guide; and (4) a Function Directory.

14. SUBJECT TERMS

Hydrodynamics
Smooth Particle Hydrodynamics
Computer Code(s)
Hydrodynamics Computer Code(s)
C Computer Language
C Computer Code

15. NUMBER OF PAGES

102

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

CONTENTS

CONTENTS (Concluded)

USER GUIDE CONTENTS

1.0 USER GUIDE

1.1 INTRODUCTION

This document is an overview of the MRC benchmark code SPHC, a smooth particle hydrodynamics code that is intended to be a testing arena for the SPH technique. The code is ANSI C compatible, and should run on any standard compiler. In addition to the routines described below, the system utilizes a variety of external functions and data-base interfaces for I/O, file maintenance, and graphics post-processing.

The SPHC package is an unusual hydro code in several respects. It is written in C and takes full advantage of the versatility and conciseness possible in this language, while maintaining a simple, FORTRAN-like style throughout. The emphasis at this stage of code development is simplicity and clarity rather than fully optimized performance. The basic algorithm is very simple and can be coded in about 100 lines of code. Unfortunately, the simplest approach requires $O(N^2)$ run time scaling, where N is the number of particles, and this is unacceptable for most serious applications. In SPHC a sophisticated tree-structured particle location technique is used to overcome this problem.

1.2 USING SPHC

SPHC has been implemented on several different environments, both PC and mainframe, and can be compiled on several different compilers as well. See the Programmers' Guide for setup procedures and allowable systems.

To run a problem on SPHC, the code module sph_init.c must be modified to initiate the desired setup conditions. For standard types of problems, setup menu code can be included in the setup routine, so a recompile is not needed for each run. The code is executed by typing "sphc" from the /sph/code directory. This results in a series of standard setup screens appearing on the terminal (see Section 6) to allow parameter adjustment. Usually, the default values will be adequate. Continued use of the "F1" key, as indicated at the bottom of the menu box, will accept all of the default values. The problem should then run, and print a step-by-step summary of the run's progress. The format of this summary is:

```
CYLINDRICAL PROBLEM

>>>STEP 0, TIME = 0, DT = 0
    KE=0.000e+000, TE=7.051e+023, PE=4.063e+022, Echeck=0.00 % / f:0
    dt_fl = 1.000e+050/0  dt_hy = 4.011e-001/16  dt_gr = 6.216e+000/85
    Error=0.00e+000, Courant mult=1.00e-001, r=0.00e+000, dt_fact=1.00e+000
        ...writing new data to /sdat/s.00...done

>>>STEP 1, TIME = 0.000e+000, DT = 1.203e-002, visc = 0.000e+000/0/0
    KE=2.380e+018, TE=7.051e+023, PE=4.063e+022, Echeck=-0.00 % / f:0
    dt_fl = 3.913e+001/17  dt_hy = 3.165e-001/16  dt_gr = 6.216e+000/85
    Error=4.98e-005, Courant mult=1.90e-001, r=4.98e-002, dt_fact=1.50e+000
```

The information shown includes: time, timestep, viscosity (ratio of the viscous pressure to the thermal pressure), kinetic, thermal and potential energies, energy check, f=number of eos faults, timesteps (list will depend on physics of the problem, includes flow, hydro, gravity, thermal diffusion), Error = the current Runge-Kutta error estimate, Courant-mult = fraction of the Courant timestep for the current dt, r = ratio of the error to the target error desired, dt_fact = the timestep adjustment factor. Also shown is the current i/o, such as files written, etc. Some quantities are followed by a slash and a particle number representing the strongest contribution. A tone will sound at the conclusion of the problem.

The code may also be initialized completely from data input. In this case an editor is used to specify the set of parameters desired in a data file, and the command form "sphc p" will read this file for values rather than using the menus, while obtaining the problem specific parameters and particle setup from the sph_init routine. Finally, a problem may be started completely from data in the form of restsrt dumps. Here a parameter file and a particle data file are required, and the form of the command is "sphc r". See Section 3 for further details.

1.2.1 Data Structures

Specialized data structures are used to organize storage. A data type "vector" is defined as three real numbers (x,y,z), and the module vect3.c contains the vector functions that form the foundation of all computations. A data structure type "particle" contains the vectors "position", "velocity", and "acceleration", as well as mass, density, temperature, etc., characterizing each particle. An array of these structures contains all the physical data. The particle structure contains two fields used to facilitate testing of tree and grid particle locator schemes: these are part_num (the particle's index number in the array), and next (a pointer to another particle, used to pass lists). Some of the needed particle data is stored in another structure: old_part. These are the particle data at the previous timestep (needed by Runge-Kutta to retry timesteps), the material number (mat_no), the thermal conductivity (thm_con), the sound speed squared (csq), and the velocity divergence times the density (rhodelv). This structure is stored in the extra segment when running on QNX, see the programmer's guide for treatment of these variables.

1.2.2 Equations

The physics of the code currently consists of the two equations

$$\frac{d^2x}{dt^2} = \text{accel} \quad , \tag{1}$$

$$\frac{de}{dt} = \text{dedt} \quad ,$$

plus the equation of state:

`p, e ( rho, T)  &  p, T ( rho, e ),`

where e is the internal energy per mass, accel the acceleration, and dedt the heating of the particle. It is the responsibility of the physics routines to add their contributions to accel and dedt.

The continuity equation is currently evaluated by computing the density as a direct sum over the mass of neighboring particles. This is the most conservative approach in terms of the numerics, and eliminates the need for explicit integration of this equation.

1.2.3 Integration

The "step" portions of the code advances the equations using a second/third order Runge-Kutta algorithm. See Fehlberg, NASA TR-R-315, 1969. The Runge-Kutta technique evaluates the maximum relative error each timestep adjusts the next timestep to achieve a desired level of accuracy. This scheme may choose integration steps larger or smaller than the usual Courant condition, as the problem warrants. The initial timestep is chosen to be one tenth the Courant step (determined from the minimum of the physics timesteps), and allowed to gradually increase. For problems with sound waves and shocks, the final timestep is usually near the physics step size. For simple translation or expansion phases, the step size is larger, but subject to a user-defined limit (generally taken to be 5 times the Courant step). For violent phases of problem evolution, such as collisions, a large step may be recalculated several times to resolve the onset of the action, and the final step may be smaller than that determined by the physics (the "physics" timesteps are estimates of the Courant stability condition for each process, in the present scheme accuracy rather than stability is used to determine the timestep).

The details of the current scheme are, given a target error "Tol", and a computed error "error" for the timestep, we have

```
          error  < 0.5 Tol      increase timestep, max = 50% increase
0.5 Tol < error  < 0.7 Tol      no change
0.7 Tol < error  < 1.5 Tol      decrease timestep
1.5 Tol < error                 redo step with decreased timestep
```

The amount of the timestep increase or decrease is computed by the Runge-Kutta code to produce the desired error adjustment. In addition, the timestep is usually limited to 5 times the normal Courant timestep as a safety margin. Finally, no increase is allowed if the previous timestep was decreased or recomputed. Tol is usually taken to be about 1.e-4.

Another option is to "turn off" the error control. In fact, this uses the above scheme as well, but also requires the timestep to be not less than the usual Courant step. Thus the timestep will be at least the usual Courant step, and the error estimate will show the resulting level of accuracy. This option is often used for a new problem to get some idea of the accuracy of a normal Courant run before setting the desired Tol.

The remainder of this note contains a description of the current architecture and options of the code, and will serve as a starting point for future additions and modifications.

1.3 FLOW CHART—OUTLINE

The logical structure of the code consists of several blocks; they are listed below showing the overall code structure. This should provide a general feel of how the computation proceeds. For details, see the Programmers' Guide.

```
Begin sphc()

    === system initialization ===

    sph_default()               set defaults

    sph_setup()                 initialize problem

    ==timestep loop==

        thermo()                compute density, equation of state,
                                conductivity, rhodelv, smoothing length

        e_compute()             compute energies, energy check

        dt_compute()            compute timestep

        run_check()             process interrupts, check if done,
                                do file i/o

        rhs()                   compute right-hand-sides of all equations
                                compute physics

        === begin new step here ===

        start_step()            store old step environment

        step1()                 Runge-Kutta advance to t + dt/2

        thermo()                compute density, equation of state
        rhs()                   compute right-hand-sides of all equations

        step2()                 Runge-Kutta advance to t + dt

    ==end timestep loop==

end sphc
```

1.4 USER OPTIONS

This code can be run in several modes. The usual method of local execution is in "interactive" mode, accessed by typing:

```
sphc
```

while in the directory /sph/code. Menus then appear that allow selection of all code parameters.

When running remotely, or if the menus are not needed, type either

```
        sph r   to execute directly from files rs and rs.par
or
        sph p   to read rs.par for parameters,
                but use sph_init() for the model setup.
```

The files rs and rs.par are normally stored in the directory "/sdat". Another directory can be selected in the main menu, and the restart files read from there. The following options are selected in the input screens, or in the "init" routines

MAIN MENU:

```
SETUP...

    Problem title:
        Problem identifier string.

    Number of particles to allocate:
        Specify the maximum number of particles desired (setup routines will
        be limited to this number).

    Spacing adjust factor:
        See "grid generator" section.

    Vary h (h_vary):
        Enable the variable smoothing length option.
```

Random Perturbation:
This percentage of random displacement in x, y and z will be applied to each particle before starting the problem. This is useful to allow the particle positions to adapt quickly to an initial distortion. This procedure can be followed by a relaxation to relieve large density fluctuations.

Particle division:
Enable subdivision of particles in regions of low resolution.

Particle combination:
Enable recombination of particles if needed.

TIME PARAMS...

Maximum time:
Normal end time for the run. If set to zero, then prompts for number of timesteps and dump interval in steps will be generated in the secondary menu.

Error tolerance timestep control:
Enable the Runge-Kutta timestep control, the relative error tolerance will be prompted for in the secondary menu. If not enabled, the timestep defaults to the "physical" timestep (minimum of Courant and other physical processes).

Timestep limit:
Maximum timestep in units of the Courant (physical) step size. Default is 5.

Run Modes:
1. Integration (default)
2. Relax - do not update velocities, but move particles
3. Relax to zero velocities - update velocities, but gradually damp all velocities to zero

Relaxation is flagged by on.relax, the zero velocity option by eq_relax. Relaxation continues until the specified time or number of steps is reached. The diagnostic "check" indicates the degree of convergence. When done, run_check resets the parameters to run mode.

PHYSICS...

Hydro (on.hydro):
Turn on the hydrodynamics, including the artificial viscosity and/or diffusion terms needed to handle shocks. PdV terms are added to the momentum and energy equations.

Thermal diffusion (on.thm_diff):
Turn on electron thermal diffusion, coefficients in eos menu.

Gravity Accel (on.grav):
This option (usually set in sph_init0) allows a constant external acceleration to be applied. The potential energy of this force is included in the energy check. If the "R" component of the gravity is set, then the gravity is directed along the radial direction (flagged by on.rad_grav).

DD/DT Thermonuclear Burn Yield (on.burn.
Turn on computation of neutron yield from DD or DT compression.

IO FLAGS...

Write data?
Flag to enable restart dumps.

Directory path:
This is the directory for restart dumps, default is /sdat.

Read data?
Read restart data from the files rs and rs.par.

Begin Again?
Make another pass through the initialization routines (start over).

SECONDARY RUN PARAMETERS

DEBUG FLAGS...

Debug Modes:
"debug" ="y" generates:
1. particle summaries for the "target" particle

2. run status information (show module executing)
3. full debug output from each executing function

If "d" is input from the keyboard during a run, toggle the debug mode.

Particle number to track:
Choose "target" particle.

Write history:
If "history" is set, a full history file for the target particle is written to /tmp/history.

Dump EOS quantities?
This includes a variety of quantities in the restart dumps that are not normally written, such as pressure, energy, etc.

GEOMETRY...

Spherical problem (on.sphere):
If dimension == 1, this option selects spherical coordinates (x is r).

Cylindrical problems (on.cylinder):
This option selects cylindrical coordinates for dimension == 1 or 2. "Y" is always the cylindrical axis (x is rho, y is z).

RUN CONTROL...

Max number of timesteps:
Displayed if max_time = 0 from first menu.

Step / Time interval for dumps:
Controls the frequency of restart/diagnostic dumps.

Accumulate dumps?
This option causes each data dump to be appended to a single dump file "s". Should be used only for small runs.

Overwrite dump file?
This option also enables a series of writes to the file "s", but here the file will be overwritten each time. If this option is NOT

selected, the dumps will be written to a series of files named "s.00", "s.01", etc. and the parameter data will be written to "s.00.par", "s.01.par", etc.

Call PLOT_MAC?
If selected, a call to the routine "plot_mac" will be attempted each time that the restart dump is updated. This routine is usually a shell file containing plotting macros.

Relative error tolerance:
Set the desired accuracy (Tol).

Relax to zero velocities?
Sets the type of relaxation.

Set gravity:
Reset the gravity acceleration vector.

Maximum / minimum particle size / mass:
Set limits to be used by particle division routines.

EQUATION OF STATE PARAMETERS MENU

Smoothing length adjustment:
This is a multiplier for the normal smoothing length that is set in the setup routines. Can be used to test the effects of longer or shorter h either at problem initialization or during a run.

Physical timestep multiplier:
This factor multiplies the Courant and other physical timestep limits to determine the actual timestep. The results are displayed during the run and used to determine the upper timestep limit.

PG gamma, poly gamma, and mu:
Parameters for the perfect gas equation of state, if selected. See the "equation of state" section.

Art-visc parameters:

Alpha	- multiplier for the linear artificial viscosity term.
beta	- multiplier for the quadratic artificial viscosity term.
g1	- multiplier for the linear artificial conductivity term.

g2 - multiplier for the quadratic artificial conductivity term.
shock density ratio - the density jump expected for the strongest shocks in the problem (used to prevent near zero timesteps).

LASER DEPOSITION MENU

This menu is activated if on.laser=TRUE is specified in sph_init0(). See the "laser deposition" section for details.

1.5 RUNTIME INTERRUPTS

The following keyboard inputs are recognized by run_check():

'?'	-	help
'p'	-	pause for input or restart
'd'	-	toggle debug output mode
'w'	-	write to file this step

The buffer is checked only one time each step, so some waiting is usually required before the keystroke is processed. Only the LAST character in the buffer is recognized, others are discarded.

1.6 RESTART CAPABILITY

At selected intervals, the problem status can be dumped to a file. The new data can either be appended to existing dumps, or replace the previous information. Two files are usually produced, both in CDAT data-base format, they are:

/sdat/s.par	-	current problem parameters
/sdat/s	-	current problem particle data.

The directory "/sdat" can be changed in the main menu. Only the independent variables and parameters are included in these files. All other needed quantities can be generated by one pass through rhs(). The number of attributes in /sdat/s varies, depending on the number of dimensions in the problem, and whether the "append" option is in effect, and whether the "dump_eos" flag is set.

It is also possible to turn off the "overwrite" flag. In this case a series of up to 99 dump files will be written, named

/sdat/s##.par	-	current problem parameters
/sdat/s##	-	current problem particle data.

where ## is 00, 01, 02, etc.

No graphics capability is included in SPHC. All needed plots can be generated from the restart dumps using available post-processors.

Optionally, a history file can be written (history = TRUE). This file contains the full status of the particle pointed to by "debug_part" for each timestep of the run. In addition a full energy history for the problem is written (kinetic, thermal, potential, laser). To suppress the particle data, set debug_part = 0.

1.7 PROBLEM INITIALIZATION

A particular problem is currently set up by modifying the routines:

```
sph_init0()          set or reset defaults and initial values
sph_init()           initialize coord, velocity, density, mass, mu,
                        for all particles
sph_bc()             appropriate boundary condition code
```

The sph_init0() call occurs after the standard system defaults are set, but before the interactive data input screens in sph_setup(). Memory allocation is based on Npart_mem set here. Npart, the actual number of particles, cannot exceed this number. Sph_init() is called after all parameters have been set and space allocated. Sph_bc() is called by "step" after the equations have been advanced each timestep, and may be used to limit particle motion in certain directions. See Section 8 for more details.

1.8 GRID GENERATOR

A standardized grid generator may be used to assist in problem initiation. Rather than coding up all particle positions in sph_init(), this routine will compute a series of x/y/z locations on a grid using a standard algorithm; it also computes the mass of each particle needed to achieve a desired initial density. Grid_gen is called from sph_init for each of possibly multiple areas of gridding with different parameters. Sph_init passes grid_gen a pointer to a function that will initialize the particles in each grid. For illustrative purposes, this function is called here "part_init". For problems with multiple blocks, a different part_init may be supplied for each block. The second argument for grid gen is an integer: "dir", which is used only for hex grids. If dir=0, one axis of the grid is aligned along the X axis and the others are at 60 degree angles. If dir=1, then the alignment is along the Y axis.

To use this option, set the following quantities in sph_init0():

```
        Grid_gen = TRUE
        Space_adjst = multiplier on grid spacing to allow for odd shapes
                    (default = 1, change from unity if the delivered number of
                    particles  does not match the desired number)

        Grid_standoff = distance to start grid from first boundary
                    (units of grid spacing, e.g =1 leaves 1 blank cell,
                    default = 0.5)

In sph_init do the following things:

    Set:
        Grid_npart = number of particles desired in current grid
        Grid_type = 1 for rectangular grid
                  = 2 for hexagonal grid
        Maxx, Minx, Maxy, Miny, Minz, Maxz: define a cube containing the
                  model
        Volume:  estimated volume of this block of the model (NOT the cube!)
        Rho:  desired density of material (g/cc)

    call:  grid_gen( "part_init", dir );     for first block

    reset parameters;

    call:  grid_gen( "part_init2", dir );    for second block

    repeat for as many blocks as desired.
```

```
In "part_init" do the following things:

        1.  test vector Grid if in desired volume, if not
            return(1);
        2.  else increment:  Part_num++
                initialize part coord = Grid,  mass = Mass
                initialize part veloc, temp, mat_no
        3.  return(1)
        4.  if any fatal errors occur, return(0)
```

Note that the generator covers the entire "cube" with grid, it is up to "part_int()" to select only the points lying in the initial model. This is usually done with a series of logical tests. The global parameter h_spacing is set equal to the particle spacing in the generator, and may be used in other functions.

The generator may be used in 1, 2, or 3 dimensions. Currently available options are:

```
    Grid_type = 1:
        Dimension = 1           Uniform grid
        Dimension = 2           Rectangular mesh
        Dimension = 3           Rectangular mesh

    Grid_type = 2:
        Dimension = 1           Uniform grid
        Dimension = 2           Hexagonal grid
        Dimension = 3           Cubic lattice of hexagonal grids
```

1.9 BOUNDARY CONDITIONS

At present, boundary conditions can be handled in two ways, selected by the value of on.boundaries:

Type 1 boundaries: on.boundaries = FALSE.

In this case the routine sph_bc() is called once each timestep after the second Runge-Kutta step, but before the "rhs" evaluation. At this time any part of the problem may be adjusted "outside" of the normal Runge-Kutta computation. Obviously, great care needs to be exercised in taking advantage of this option. The usual application is that of reflecting particles at a wall. The code looks like this, assuming that the xmin_bc, etc. have been set in sph_init0():

```
sph_bc()    /*------------------------ handle any boundary conditions  */
{
    int i;
                                                /*  three sided box  */
                                                /*  reflecting sides */

    for( i = 1; i <= Npart; i++ ) {
        if( part[i].coord.x < xmin_bc ) {
            part[i].coord.x = xmin_bc;
            part[i].veloc.x *= -1.;
        }
        if( part[i].coord.x > xmax_bc ) {
            part[i].coord.x = xmax_bc;
            part[i].veloc.x *= -1.;
        }
        if( part[i].coord.y > ymax_bc ) {
            part[i].coord.y = ymax_bc;
            part[i].veloc.y *= -1.;
        }
    }
}
```

Type 2 boundaries: on.boundaries = TRUE:

This option turns on a general boundary condition handler. The number, type, and other characteristics of the boundaries are set in sph_init0, as in the following example:

```
                                                /*  boundaries  */
    on.boundaries = TRUE;

    bounds[1].type = PER;
    bounds[1].per_type = X2;
    bounds[1].per_offset.x = -size;

    bounds[2].type = PER;
    bounds[2].per_type = X2;
    bounds[2].per_offset.x = size;

    bounds[3].type = REFL;
```

In this case three boundaries are specified, boundaries 1 and 2 are periodic, while boundary 3 is reflecting. Valid boundary options are: WALL, REFL (reflecting), PER (periodic), AXIS, FLOW (flow-thru), FIXED, and PISTON. The last three are not yet implemented. In the case of PER boundaries, the type can be X2, Y2, or Z2 for fixed offsets in x, y, or z directions, and the offset is specified as the distance from the other boundary (in the above example, boundary 1 is at 0, boundary 2 is at "size"). Care should be taken to insure that these values are consistent with the boundaries specified in sph_bc().

In sph_bc() the user is obliged to supply code to test whether a particle is within 2*h of the boundary and return information about it, namely, the number of boundaries near the particle, which boundaries they are, the coordinates of the nearest points on each boundary, and a flag that indicates whether the particle is on the wrong side of any of the boundaries. The code below illustrates this.

```
/*--------------------------- BOUNDARIES ----------------------------------*/
/*  sph_bc will be called for each particle if on.boundaries == TRUE  */
/*
    arguments:
            coord           =   position of particle
            h               =   smoothing length of particle
    return:
            numb            =   number of boundaries within 2h
                                must be <= 5
            num             =   boundary number of each
            bdry[1->numb]   =   location of closest point on each boundary
                                plus boundary number
                                plus crossed = TRUE is particle has crossed
*/


sph_bc( coord, h, numb, bdry )  /*---------------------------------------*/
vect_ptr coord;
struct bdry_list bdry[];
float h;
int *numb;
{
                                             /*  fixed boundaries          */
    *numb = 0;
                                             /*  axis  */
    if( coord->x < 2. * h ) {
        (*numb)++;
        bdry[*numb].coord.y = coord->y;
        bdry[*numb].coord.x = 0.;
        bdry[*numb].coord.z = 0.;
        bdry[*numb].num = 1;
        if( coord->x < 0. )  bdry[*numb].crossed = TRUE;
        else                 bdry[*numb].crossed = FALSE;
    }

                                          /*  outer boundary   */
    if( coord->x > xmax_bc - 2. * h ) {
        (*numb)++;
        bdry[*numb].coord.y = coord->y;
        bdry[*numb].coord.x = xmax_bc;
        bdry[*numb].coord.z = 0.;
        bdry[*numb].num = 2;
        if( coord->x > xmax_bc )  bdry[*numb].crossed = TRUE;
        else                      bdry[*numb].crossed = FALSE;
    }

                                          /*  lower boundary  */
    if( coord->y <  2. * h ) {
        (*numb)++;
        bdry[*numb].coord.x = coord->x;
        bdry[*numb].coord.y = 0.;
        bdry[*numb].coord.z = 0.;
        bdry[*numb].num = 3;
        if( coord->y < 0. )  bdry[*numb].crossed = TRUE;
        else                 bdry[*numb].crossed = FALSE;
    }

    return(1);
}
```

Warning: note that the bdry and coord vectors are referred to differently in sph_bc, we put bdry[i].x, but use coord->x. This usage is forced because we pass a pointer to the coord vector structure (as in the vect3 routines), but the "bdry" in the calling sequence is a pointer to an ARRAY of vector structures (note that an & is not needed on bdry when calling this routine, since array names are treated as pointers, as in fortran). The declarations in sph_bc make this clear.

See the SPH Technical Guide for details on boundary implementation.

1.10 INPUT MENU

Any particular problem will have its own set of parameters that define the various sizes, shapes, velocities, masses, densities, etc., of the components of the problem. These are set in sph_init0() or in sph_init(). Often, many of these parameters will need to be varied during the course of several problem runs. It is convenient to code a simple menu in sph_init0() to allow run-time adjustment of these parameters. The following example shows how this is done.

```
                                                      /*  problem  */
    strcpy( cool_name, "ch" );
    strcpy( hot_name, "ch.1" );
    hot_temp = 1.e5;
    cool_temp = 1.e3;
    hot_den = 0.1;
    cool_den = 0.1;
    height = .005;
    width = 0.02;
    groove_depth = 0.2;                                /*  fract of height  */
    groove_width = 0.25;         /*  fract of width (full wavelength)  */

    line = 1;
    reset_param();
    set_param( "F1 = continue, ESC = exit", "control", 0, 0, 0 );
    set_param( "LA2_2D PARAMS", "header", 0, 0, 0 );
    set_param( "Target width (cm)", "double", &width, 0, 0 );
    set_param( "Target height (cm)", "double", &height, 0, 0 );
    set_param( "Hot Material", "string", &hot_name, 0, 11 );
    set_param( "Hot Temp (K)", "double", &hot_temp, 0, 0 );
    set_param( "Hot Density / solid", "double", &hot_den, 0, 0 );
    set_param( "Cool Material", "string", &cool_name, 0, 11 );
    set_param( "Cool Temp (K)", "double", &cool_temp, 0, 0 );
    set_param( "Cool Density / solid", "double", &cool_den, 0, 0 );
    set_param( "Groove fract depth", "double", &groove_depth, 0, 0 );
    set_param( "Groove fract wavelength", "double", &groove_width, 0, 0 );
menu:
    if( interactive) {
        ret_val = edit_param( line );
        if( !ret_val )  return( 0 );
    }
    cool_mat = mat_lookup( cool_name );
    hot_mat = mat_lookup( hot_name );
    if( interactive) {
        if( cool_mat == -1 ) {
            line = 6;
            goto menu;
        }
        if( hot_mat == -1 ) {
            line = 3;
            goto menu;
        }
    }
```

1.11 EQUATION OF STATE

The implementation of the analytic equation of state is described in detail in the SPH Technical Manual. Here we outline the method of accessing this routine in the problem setup.

The example in the previous section shows a problem with two materials. For convenience, the routine mat_lookup(name) has been included in the module eos.c . This routine takes a material name as an argument, and returns the table entry number for that material. Valid materials and parameters are listed in Table 1.

A return value of -1 indicates failure to match the label. The table number of each material is included in the restart dump.

The first entry (mat_no = 0) is "pg", indicating a perfect gas equation of state. The equations used here are

```
P = (R/Mu) rho * T,
```

and

```
T = (R/Mu) * (Gamma - 1) * E.
```

The parameters Gamma (ratio of specific heats), and Mu (mean molecular weight), are set in the setup portion of the SPHC system. An additional parameter, GammaP (polytropic gamma) must also be specified. This parameter can be used to specify a fixed

TABLE 1.

```
/*  material list    0     1     2     3       4     5     6     7      8     9     10  */
char *matname[] = { "pg", "ch", "al", "sio2", "n2", "au", "dt", "h2o", "pb", "h2", "d2", "" };
                                                   /*  Grueneissen gamma(-1) */
double gamma_G[] = { .6667, 1.18, 2.136, 0.60, 0.4, 2.225, 2., 2., 2., 2., 2. };
                                                   /*  gamma for liquid  */
double gamma_liq[] = { 1.6667, 1.1, 1.4, 1.1, 1.4, 1.4, 1.4, 1.2, 1.4, 1.4, 1.4 };
                                                   /*  molecular gamma  */
double gamma_mol[] = { 1.6667, 1.4, 1.6667, 1.3333, 1.4, 1.6667, 1.4, 1.2, 1.666667, 1.4, 1.4 };
                                                   /*  solid den and c  */
double rho_0[] = { 1., 1.044, 2.7, 2.204, 1.3, 19.32, 0.30, 0.9982, 11.4, .088, .176 };
double csq_0[] = { 1., 6.25e10, 4.12e11, 3.6e11, 1.e11, 1.05e11, 1.e11, 2.41e10, 4.67e10, 1.e11, 1.e10 };
                                                   /*  liquid c  */
double csq_1[] = { 0., 1.0e10, 4.0e9, 1.e9, 3.e8, 1.05e9, 1.65e8, 1.e9, 4.0e9, 1.65e8, 1.65e8 };
                                                   /*  molecular at wt  */
double a_mol[] = { 2., 13., 27., 60., 28., 197., 5., 18., 207., 2., 4. };
                                                   /*  mean atom wt & chg  */
double a_atm[] = { 2., 6.5, 27., 20., 14., 197., 2.5, 6., 207., 1., 2. };
double z_atm[] = { 1., 3.5, 13., 10., 7., 79., 1., 3.3333, 82., 1., 1. };
                                                   /*  vaporization temp & enrgy */
double tvap[] = { 0., 3500., 5000., 10000., 77., 5000., 20., 373., 2023., 20., 20. };
double hvap[] = { 0., 5.59e9, 1.155e11, 1.53e9, 2.24e9, 4.45e9, 0., 2.82e10, 8.83e9, 0., 0. };
                                                   /*  dissociation temp & enrgy */
double tdiss[] = { 0., 3500., 0., 10000, 11000., 0., 20., 3000., 0., 20., 20. };
double hdiss[] = { 0., 2.56e11, 0., 1.28e11, 3.34e11, 0., 0., 2.66e11, 0., 0., 0. };
                                                   /*  triple point temperature */
                                                   /*  force pg above here  */
double tcp[] = { 0., 13860., 5726., 10000., 1000., 8460., 33., 650., 5223., 33., 33. };
                                                   /*  tensile strength  */
double pten[] = { 1.e9, 1.e9, 5.3e11, 1.e10, 1.e9, 3.e11, 1.e9, 1.e10, 4.6e10, 1.e9, 1.e9 };
```

constant dQ/dT to generate "polytropic" evolution. GammaP = Gamma results in the adiabatic case, while GammaP = 1 produces an isothermal model. See the Technical Manual for details and other examples. These three parameters are not used if no perfect gas regions are specified, and all perfect gas regions are constrained to use the same parameters with the present setup.

In addition, the code provides the capability of distinguishing between several regions of the same material. It does this by allowing 100, 200, etc. to be added to a material number with no effect on the eos computations. Thus three regions of dt may be labeled 6, 106 and 206. All use the same eos parameters. The different materials are labeled "dt", "dt.1", and "dt.2" in the call to mat_lookup(). When accessing eos data directly (using the solid densities, for example), be sure to use hot_mat%100 as the index to split off the "100" tag.

When initializing the particle array, the material number is stored in the extra segment as @&old_part[i].mat_no and this is an integer member of the old_particle structure. This parameter is stored in old_part simply to save space. It is a current value, NOT an "old" value. In fact, this data is never expected to change for ordinary problems.

1.12 LASER DEPOSITION

Laser deposition is selected by setting on_laser = TRUE. The following parameters must then be set (this is usually done in laser_setup() via a pop-up menu):

`E_laser`	`laser energy, Joules`
`Fwhm`	`pulse length, ns`
`Spot_rad`	`laser spot radius, microns`
`Wavelength`	`laser wavelength, microns`
`F_number`	`laser f-number (def=6)` `the convergence half-angle is 1/2F radians`
`Bremfac`	`inverse Bremsstrahlung multiplier (def=0.5)`
`NfeO`	`fraction pre-ionization at 1 cm from target (def=.05)` `(fraction of singly ionized ions initially present)`

These parameters plus laser_etot, the total energy currently deposited via laser deposition are written to restart dumps. All parameters are converted to cgs units immediately after entry.

When setting up problems with laser deposition, observe the following conventions:

> The laser propagates from infinity along the Y axis.
> The initial target surface is located at Y = 0.

1.13 PARTICLE DIVISION/CREATION

Version "bl" includes a test version of a new particle division algorithm. The intent here is to increase resolution in low density areas of a problem by dividing particles whose smoothing length exceeds some threshold. To use this option, set the following parameters:

> on.divide = TRUE
> max_h = smoothing length threshold for particle division
> min_mass = minimum mass allowed (this limits the division process)

Obviously, this option will only work if variable smoothing length is used.

In other problems, the addition or subtraction of particles at boundaries may be desirable. Inflow and outflow boundary conditions (not implemented at present) would be an example. If the details in a certain region of the problem are not of interest, the particles can be dropped from the calculation in the following way:

```
    Set    on.delete = TRUE

    In sph_bc, include code of the form:

                                                              /*  move top
boundary  */
     if( h == 0. ) {
          ymax_bc = NEW LOCATION OF BOUNDARY
          if( pcoord->y > ymax_bc ) {
               return( -1 );
          }
          else  return(1);
     }
```

A special call is made to sph_bc() with h = 0 from drop_particles(), which is called by thermo() if on.delete = TRUE. If a value of -1 is returned, the particle is dropped. This call is made before the density or equation of state is computed.

PROGRAMMER'S GUIDE CONTENTS

2.0 PROGRAMMER'S GUIDE

2.1 INTRODUCTION

This is a supplement to the SPHC User's Guide, and is intended to supply the information needed to perform modifications to the code. It also sets needed guidelines for code format and organization.

2.2 FLOW CHART—DETAILS

The chart shown below is a more detailed flow/structure chart for the code. Here all routine calls are shown in the order of execution, with indentation to indicate dependencies. Also, a brief description of the purpose of each routine is given. See the User_guide for a summary of this chart. See the Function directory for details of each routine.

```
START:
    ---system inits---
        set_priority()          set for low priority run
        process arguments       check for restart conditions
        initscr() / scrollok()  set up menu system flags


    sph_default()               set defaults
        .....                   set all default values
        ker_setup()             set kernel parameters
        sph_init0()             problem dependent routine
                                to set or reset parameters
        laser_setup()           set laser params (if on.laser)


    sph_setup()                 initialize problem
        screen input routines:
             reset_param, set_param, edit_param

        read_param()            read problem parameters
        calloc()                allocate memory
        alloc_segment()         allocate extra segment memory
        read_dump()             read restart file
        eos_init()              set eos and art visc params (input)
            screen input routines
        sph_init()              problem setup (problem dependent)
            grid_gen()          optional, repeat possible
                "part_init()"   user specified particle setup routines
            smooth_temp()       optional smoothing step for initial temps
        sph_random()            apply random perturbations to x, y, z
            adjust masses       for curvilinear problems
        file_init()             open restart file, open history file

        nn_compute()            set initial smoothing lengths to nearest
                                neighbor  distances

TIMESTEP LOOP:

    check_boundaries()          check for illegal particle positions
        sph_bc()                user boundary routine
        reflect_particle()      reposition wayward particles
            sph_bc()
            vector_routines

    init_nbr()                  initialize neighbor list pointers

    thermo()
        .....                   zero thm_con, av_mu_max

        drop_particles()        drop unneeded particles

        h_compute()             update smoothing length

        rho_compute()           compute density
            get_nbr_list()      find neighbors
                miss()          check particle distances
                octree()        alternate tree scheme
            do_rho()            compute i/j density terms
                kernel()        compute Wij
                vector routines
            do_bounds()         compute boundary density terms
                do_rho()        used for ghost boundaries
                rho_wall()      used for wall boundaries
                vector routines
    divide_particles()      divide particles in low density areas

    eos_compute()
        eos_pe              compute pressure and energy (step 1)
        eos_pt              compute pressure and temperature (steps > 1 )

    e_diff()                compute thermal diffusion

e_compute()                 compute energies, energy check

dt_compute()                compute physics (Courant) timesteps
    .....                   compute error control timestep
    .....                   retry timestep if needed

run_check()                 process interrupts, check if done
    char_waiting()            check for keyboard input
    write_part()              debug output
    write_step()              restart dump
        file_init()             re-open file if needed
        write_param()           write parameter summary to disk
        plot_mac                generate graphics output
    ....                      display run-time information

rhs()                       compute right-hand-sides of all equations

    .....                   zero accel, dedt


    gravity()               compute accel due to gravity
        vector routines

    hydro()                 compute accel due to hydro
        get_nbr_list()        find neighbors
        vector routines
        do_hydro()            particle i/j force terms
            art_visc()          artificial viscosity
            kernel()            kernel, for force computation
            vector routines
        do_bounds()           boundary forces
            hydro_wall()        wall force terms
```

```
                    art_visc()        artificial viscosity
                    kernel()          kernel, for boundary forces
                    vector routines
                do_hydro()        ghost particle force terms
                vector routines

        thm_diff()                compute thermal, viscous, and radiation diff
            get_nbr_list          find neighbors
            vector routines
            kernel                kernel, for diffusion equation

        burn_yield()              calculate neutron production for DD/DT burn
                                      (if on.burn)

        laser_dep()               compute laser deposition (if on.laser)

        poly()                    apply polytropic corrections

    <NEW TIMESTEP BEGINS HERE>

    start_step()
        .....                     print step header
        vector routines           store old values and rhs's

    step1()                       first RK step
        vector routines           advance equations to t + dt/2

    sph_bc()                      do boundary conditions (problem dependent)

    thermo()                      see above
    rhs()                         see above

    step2()                       second RK step
        vector routines           use old variables and rhs's
        .....                     advance equations to t + dt

    sph_bc()                      do boundary conditions (problem dependent)

\clearpage

    cleanup_nbr()                 clean up neighbor list pointers
```

```
END TIMESTEP LOOP
END


DETAILS OF BOUNDARY TREATMENT

physics routine (hydro, etc..)
    do_bounds()             driver
        sph_bc()            user routine to check for boundaries

        *do_wall()          pointer to WALL boundary routine

        get_refl_list()     prepare REFLECT pointers
            get_nbr_list()
            vector routines
        do_refl()           REFLECT boundary
            copy_particle()
            reflect_particle()
                sph_bc()
                vector_routines
            *do_ghost()     pointer to ghost handler

        get_per_list()      prepare PERIODIC pointers
            get_nbr_list()
            vector routines
        do_per()            PERIODIC boundary
            copy_particle()
            *do_ghost()     pointer to ghost handler
```

Here, *do_wall and *do_ghost are pointers to physics routines to handle the interactions with walls and ghost particles. They are passed in the calling sequence of do_bounds. Boundaries in addition to the basic WALL, REFLECT, and PERIODIC will be added in the future. For further details, see the comments in each routine.

2.3 SPHLIB DIRECTORY (routine locations)

The indented names are routine names, the ".o" names are the file names. For example, the source for "eos_pe()" is found in the file "eos.c".

```
burn.o
    burn_yield
    burn_part
    sigmav_bar
laser.o
    laser_setup
    laser_spot
    laser_dep
    laser_pulse
    bremk
step.o
    start_step
    step1
    step2
thm_diff.o
    thm_diff
hydro.o
    hydro
    do_hydro
    hydro_wall
    gravity
sph_bounds.o
    do_bounds
    get_refl_list
    get_per_list
    do_refl
    do_per
    check_boundaries
    reflect particle
eos.o
    eos_init
    eos_pe
    nrg
    eos_pt
    t_nrg
    mat_lookup
    art_visc
    poly
kernel.o
    ker_setup
    kernel
    norm
SPHCompute.o
    eos_compute
    dt_compute
    nn_compute
    h_compute
    e_compute
rho.o
    rho_compute
    do_rho
    rho_wall
run_check.o
    run_check
    finish
sph_utl.o
    sph_random
    nfel
    e_diff
    smooth_temp
    divide_particles
    divide_part[illegible]
    copy particle
    delete_particle
    drop_particles
vect3.o
    vzero
    veq
    vsum
    vdiff
    vav
    vmult
    vlin
    dot_prod
    cross_prod
    vect_len
    vect_len2
    x_veq
    x2_veq
    x_vlin
    x_vdiff
```

The particle neighbor routines are machine dependent (e.g., "miss" is used for small machines, "oct-tree" for large machines, etc.) they are linked separately. They currently include:

```
miss.o
    get_nbr_list
    init_nbr
    cleanup_nbr
    miss

octree.o
    get_nbr_list
    init_nbr
    cleanup_nbr
    octree
    alloc_root
    dump_tree
    free_tree
    build_particle_list
    split_node
    do_tree
```

For the PC version of the code, a second code segment is defined, called SEG2 in the Makefile, it includes:

```
sph_default.o
    sph_default
sph_setup.o
    sph_setup
sph_io.o
    file_init
    write_step
    read_dump
    write_param
    read_param
    write_part
grid_gen.o
    grid_gen
```

2.4 GLOBAL DEFINITIONS

All globally defined variables, type definitions and "defines" are found in the file "sph.h", listed here:

```
/*  sph.h -- header file for SPHC routines  */
/*  version  bk  11/08/89  -  Copyright (C), Mission Research Corp., 1989  */

#include <stdio.h>
#include <math.h>
#include <const.h>

/*=====================================================================*/
                                                  /*  systems available  */
#define QNXX        0  /*  QNX 286 compiler  */
#define QNXY        1  /*  C86 compiler  */
#define UNICOS      2
#define MSDOS       3
                                                  /*  type of pop-up menus  */
#define CURSES      0
#define VTERM       1
                                                  /*  set up system here  */
#define SYSTEM       QNXX
#define MENULIB      VTERM

                                                  /*  menu info  */
#if SYSTEM == QNXX
#if MENULIB == CURSES
#define printf        printw
#define fprintw       fprintf
#endif
#endif
                                                  /*  system file info  */
#if SYSTEM == MSDOS
#define PATH       "\sdat\"
#define MODR       "r+"
#define MODW       "w+"
#define MODA       "a+"
#else
#define PATH       "/sdat/"
#define MODR       "rv"
```

```
#define MODW       "wv"
#define MODA       "av"
#endif
                                                  /*  memory limit/k  */
#define Kmem        64

                                        /* Defines for Ansi C Compatibility */
#if SYSTEM == QNXX
#define EX_S @
#define EX_P -}
#else
#define EX_S *
#define EX_P ->
#endif

/*==========================================================================*/

                                                  /*  handy macros  */
#define sq( x )         ((x) * (x))
#define cub( x )        ((x) * (x) * (x))
#define cbrt( x )        (pow( (x), (1./3.) ))
#define quad( x )       ((x) * (x) * (x) * (x))
#define fmin( x, y )    ((x) < (y) ? (x) : (y))
#define fmax( x, y )    ((x) > (y) ? (x) : (y))

                                                  /*  constants  */
#define R          8.3170e+07  /*  erg/deg/mole  */
#define K          1.3805e-16  /*  erg/deg  */
#define Kev        1.6022e-12  /*  erg/eV   */
#define SIG        5.6690e-05  /*  erg/cm2/deg^4/sec  */

                                                  /*  logic  */
#define LOGICAL    int
#define TRUE       1
#define FALSE      0

                                                  /*  grid types  */
#define RECT       1
#define HEX        2

                                                  /* boundaries */
```

```
#define MAX_BOUNDARIES   6
                                                /* boundary types  */
#define WALL   1
#define PISTON 2
#define PER    3
#define REFL   4
#define FLOW   5
#define AXIS   6
#define FIXED  7

                            /* Periodic types - several can be on at once */
#define  X2  1
#define  Y2  2
#define  Z2  4

/*==============  data structure defs ====================================  */

                                                /*  run flags  */
struct run_params {
    LOGICAL sphere;                 /*  spherical coords  */
    LOGICAL cylinder;               /*  cylindrical coords  */
    LOGICAL boundaries;             /*  boundary conditions in effect  */
    LOGICAL relax;                  /*  relax to quiet start  */
    LOGICAL hydro;                  /*  do hydro  */
    LOGICAL grav;                   /*  turn on gravity  */
    LOGICAL rad_grav;               /*  turn on radial gravity  */
    LOGICAL av_diff;                /*  turn on viscous thermal diffusion  */
    LOGICAL elect_diff;             /*  turn on electron thermal diffusion  */
    LOGICAL rad_diff;               /*  turn on radiation diffusion  */
    LOGICAL laser;                  /*  turn on laser deposition  */
    LOGICAL burn;                   /*  turn on DT/DD burn neutron production
*/
    LOGICAL divide;                 /*  turn on particle division  */
    LOGICAL combine;                /*  turn on particle recombination  */
    LOGICAL create;                 /*  turn on particle creation  */
    LOGICAL delete;                 /*  turn on particle deletion  */
};
                                                /*  3D vector  */
struct vector {
    float x;
    float y;
    float z;
};
                                                       /*  vector pointer type  */
typedef struct vector *vect_ptr;
                                                       /*  functions  */
extern double vect_len(), vect_len2(), dot_prod();

                                                       /*  kernel parameters  */
struct kerdat {
    char name[15];
    double opt_space;                        /*  ideal spacing in terms of h  */
    double cutoff;                           /*  h at which kernel is effectively 0  */
};
                                                       /*  problem identifiers  */
struct titles {
    char driver[15];
    char problem[15];
};
                                                       /*  define a particle  */
struct particle {
    struct vector coord;
    struct vector veloc;
    struct vector accel;
    float energy, dedt;

    float mass;
    float sm_ln, temp, density, pressure;

    int part_num;
    struct particle *next;
                                                       /*  scratch pointer  */
    struct particle *nxxt;
};
                                                       /*  previous values  */
                                                       /*  to put in extra seg  */
struct old_particle {
    int mat_no;
    float rhodelv, thm_con, csq;

    struct vector o_coord;
    struct vector o_veloc;
    struct vector o_accel;
    float o_energy;
    float o_dedt;
    float o_density;
    float o_sm_ln;
};

struct bdry_list {
    struct vector coord;
    int num;
    int crossed; /* TRUE - particle is on the wrong side of boundary */
                 /* will be returned to problem space according to boundary
                    conditions by check_bounds */
};

struct bound {
    int type;
    /* for Periodic only */
    /* Periodic boundaries must be two parallel boundaries in X,Y, or Z
       No checking is performed to verify consistency,
       the offset is subtracted from the appropriate boundary */
    int per_type; /* 2X, 2Y, 2Z */
    struct vector per_offset;
};

struct timestep {
    char name[15];
    int part;
    double dt;
};

/*============== common variables here ================================= */

                                                    /* general parameters */
EXTERN struct run_params on;
EXTERN struct particle *part;
EXTERN struct old_particle *old_part;
EXTERN struct vector grav;
EXTERN double grav_r;
EXTERN LOGICAL append, overwrite, history, debug, dump_eos;
EXTERN LOGICAL plot_macro, dt_cntrl, interactive;
EXTERN int dimension, Npart_mem;
EXTERN int Npart, istep, max_steps, step_skip, debug_part, plot_num,
dump_number;
EXTERN double e_time, dt, max_time, time_skip, timestep_const;
EXTERN double pert_size, err_tol;
EXTERN double k_e, t_e, p_e, en_tot, en0;
EXTERN char path_name[81];

                                                  /*  for sph_utl  */
EXTERN struct titles Titles;
                                                  /*  for eos  */
EXTERN double Gamma, GammaP, Mu;
EXTERN double rho_solid, c_solid;
EXTERN double av_alpha, av_beta, av_den_ratio, av_mu_max;
EXTERN double av_g1, av_g2;
EXTERN int av_i, av_j;

                                                  /*  timesteps  */
EXTERN double dt_phys, dtold, cou_limit;
EXTERN struct timestep dthyd;
EXTERN struct timestep dtflow;
EXTERN struct timestep dtgrav;
EXTERN struct timestep dtthm;
EXTERN int retry;
EXTERN LOGICAL first_half_step;

                                                  /*  for grid generator  */
EXTERN LOGICAL Grid_gen;
EXTERN int Part_num, Grid_npart, Grid_type;
EXTERN double Volume, Rho, Maxx, Minx, Maxy, Miny, Maxz, Minz, Space_adjst;
EXTERN double Mass, Grid_standoff;
EXTERN struct vector Grid;

                                                  /*  boundaries  */
EXTERN struct bdry_list bd_list[MAX_BOUNDARIES+1];
EXTERN struct bound bounds[MAX_BOUNDARIES+1];

                                                  /*  for kernel  */
EXTERN struct kerdat Ker;
EXTERN double h_inp, h_spacing;
EXTERN LOGICAL h_vary;
```

```
                                                          /*  for thm_diff  */
EXTERN double Flux_lim, Thm_mult;
                                                          /*  for laser  */
EXTERN double E_laser, Fwhm, Spot_rad, Wavelength, F_number, Bremfac, NfeO;
EXTERN double laser_pwr, laser_energy, laser_etot, laser_utot;

                                                          /*  for DT/DD burn */
EXTERN double total_neutrons;

                                                          /*  for divide / combine  */
EXTERN double max_h, min_mass, min_h, max_mass;

/*==================== temporary variables here ========================= */

EXTERN int cum_retries, energy_faults, ghost_part_num;
EXTERN LOGICAL reset_energy, eq_relax;
EXTERN LOGICAL begin_prob, read_file, read_data, write_file;
EXTERN double time0, conv_check, max_cou_mult;
EXTERN char start_time[25];
```

2.5 CODE UPDATE GUIDE

This is a checklist of things to do when updating the code or adding options. To simplify the code structure, SPHC requires that pieces be added to existing routines for changes of this type. If the new piece of code is large, consider inserting calls to new routines (as done e.g., in eos_init / eos, and sph_init0 / sph_init). As a standard example, we consider the coding needed to add a new piece of physics to the code.

```
todo                                      code module
-------------------------------------------------------------------

write new_physics()                       new_physics()
    add to directory and Makefile         directory, Makefile
    add new_physics_flag to "run_params"  sph.h
        initialize flag                   sph_default()
    add call in rhs()                     sphc.o
    if new equation, update RK scheme     step.o
    update this manual                    Prog_guide
                                          User_guide
                                          Tech_guide
```

```
                                        Func_dir

optional changes

    add new_physics global variables    sph.h
        set defaults                    sph_default()
        interactive variable reset      sph_setup()
        computed quantities             SPHCompute.o
        add variables to restart dump   sph_utl routines

    add new_physics timestep            SPHCompute()
        add to timestep structure       sph.h
        add to timestep display         SPHCompute()
        set timestep name               sph_default()
-------------------------------------------------------------------
```

Within each routine be sure that the new code is separate and clearly labeled as to function, date of addition, and programmer (unlabeled means Stellingwerf). Also be sure to conform to code structure and variable names used for other modules. If you don't like the existing structure or names, please contact the authors of the affected sections with your suggestions.

2.6 CODE STYLE

As a matter of clarity and consistency, we use a coding style similar to the classical Kernighan and Ritchie format.

```
    The basic rules are:

        1.  Indent four spaces at every "{".
        2.  Exdent four spaces at every "}".
        3.  Do not indent or exdent in ANY other circumstance.
            (Indentation is used to keep track of brackets.  The exact
            placement of brackets is up to the programmer).

        4.  Labels (used sparingly) are placed at the left margin.
label:
        5.  Short comments are normally on a blank line and offset right.

                                                  /*  a short comment  */
```

6. Place longer comments flush right in the code.

```
                                /*  this is an example of a longer comment  */
```

..or completely across if a section is delimited.

```
/*---------------------- a major section comment -----------------------*/
```

7. Use blank lines liberally to enhance readability.

8. Use standard FORTRAN style statements whenever possible, avoid lots of imbedded or unreadable C constructs, even if more efficient.

9. Tabs may be used, but only for indentation at left margin. Use the QNX convention: TAB = [4 spaces] whenever possible.

Note that nested ifs and fors do not require brackets in C if only a single statement is in their range. PUT IN the brackets anyway to enhance readability and avoid the future shock when someone adds a statement or two.

Note that switches are sometimes indented on the colon. The above rules forbid this unless an extra set of brackets are added for each option in the switch.

2.7 UPDATING THE FUNCTION DIRECTORY

When adding new functions add a brief description to the directory in the format (examples included):

```
function/module name()

    module:         xxx.o
    function:       yyy.c, zzz.c

    type:           void

    arguments:
        int kk      first one
        double ff   second one
```

```
calls:          vector routines, etc.

principal global variables:
    first       the first one
    second      the second one

principal local variables:
    a           the local value of a
    b           the local value of b
    other       not important

description:

Include here a brief statement of the routine's purpose in life.
Describe any cute algorithms and explain coding that may be obscure.

references:

List any references that may be useful, including other routines and
programs.  Include specific references for algorithms.
```

2.8 SYSTEM IMPLEMENTATION

The SPHC code is written to be compatible across several different operating systems with few code changes. Most code changes are made in the header file sph.h. These changes are addressed in the section below pertaining to the particular operation system used. To facilitate program input, SPHC is linked to a CURSES library. The CRAY and QNX systems have a system dependant CURSES library while a CURSES subset library is provide with the DOS version. The following sections describe the code changes necessary for each particular operating system.

CRAY OPERATION SYSTEM VERSION

In sph.h, for CRAY operation under the UNICOS operating system, include the definition:

```
#define SYSTEM UNICOS
```

This will automatically install the changes necessary for CRAY operation.

QNX OPERATION SYSTEM VERSION

For the QNX version of SPHC the variable defined as SYSTEM in sph.h must be set to either QNXX or QNXY depending on the compiler used. QNXX for the standard QNX compiler and QNXY for the QNX version of the Computer Innovations, INC. Optimizing C86 Compiler. SPHC may also be linked to either the CURSES or standard video interface libraries. The variable defined as MENULIB in sph.h must be set to either CURSES or VTERM to link to these libraries respectively. The parameters changed by defining SYSTEM and MENULIB deal with file I/O, default output directory path, and extra memory segment addressing peculiar to the QNX operating system.

DOS OPERATING SYSTEM VERSION

For the DOS version of SPHC the variable defined as SYSTEM in SPH.H must be set MSDOS. This version is presently set up to compile with the Computer Innovations, INC. Optimizing C86 Compiler. SPHC may also be linked only to the CURSES video interface library. The variable defined as MENULIB in SPH.H must be set to CURSES to link to this libraries. The parameters changed by defining SYSTEM and MENULIB deal with file I/O and default output directory path.

(a). Setting up DOS system for SPHC usage

If you are starting from installation of SPHC on a DOS operating system this and the following sub-section will be useful.

First, your CONFIG.SYS file must contain the following definitions:

```
DEVICE=(path)\VDISK.SYS 128 512 /E
DEVICE=(path)\ANSI.SYS
FILES=30
BUFFERS=30 (=7 under multitasking DOS)
FCBS=32,32
```

where "path" is the directory path to your RAM disk driver and ANSI control character interpreter driver (it's best if they are in the same directory). The FILES, BUFFERS and FCBS definitions are necessary for the C86 compiler mentioned above. Multitasking DOS handles file caching much better than regular DOS buffering which is why only the default number of buffers is necessary.

Second, your AUTOEXEC.BAT file must contain the following lines:

```
PATH=(your path declarations);c:\c86
C86TEMP=d: (or e: if this is your defined RAM disk)
```

where PATH is the default system search path and C86TEMP is a C86 compiler variable defined as the location for the temporary files generated while compiling SPH. Use of a RAM disk is highly recommended to reduce hard disk access therefore shortening compilation time. It is also recommend that you have you C compiler (C86) on you hard drive (which should be drive C) in the directory \c86. Adding "c:\c86" to your default PATH definition makes compilation from the SPHC directory more convenient.

(b). Setting up SPHC directories

The set up of the SPHC and related routines under DOS should be as follows. First, you should have a directory called \sph\code (similar to that under the QNX operating system). This directory needs three sub-directories: \sphlib, \curlib, and \plib. The directory \sph\code contains the SPHC driver source and the SPH_INIT input codes, SPHC.C and SPH_INIT.C respectively. The SPH executable will also reside in this directory. Provided with the distribution copy of SPH-DOS is a DOS batch file called MAKE.BAT. This routine is provided to aid in building the SPHC executable. Its' usage will be discussed later.

The sub-directory \sphlib contains all the physics subroutines called by SPHC.C and SPH_INIT.C. The file DUMMY2.C contains certain routines peculiar to the DOS operating system, those found in the QNX compiler but not in the C86 compiler.

The sub-directory \curlib contains all the routines necessary for terminal (screen) I/O. These routines produce all the pop-up menus seen when first running SPH. There is also a file called MAKE.BAT. This is a DOS batch file which, when executed, will create the CURSES library called CURLIB.LIB from all the source code in this directory.

The sub-directory \plib contains all the actual low level screen interface routines. These make use of the ANSI.SYS definitions. The file called MAKE.BAT in this directory creates the library called PLIB.LIB that contains the object code compiled from the source routines in this directory.

In addition the directories \sdat and \sph\tests must be created. These contain respectively, the output generated while executing SPHC and some test input SPH_INIT.C files.

There are also several header files that must be copied into the directory \c86 from the distribution disk directory by the same name, \c86.

(c). Creating and running SPH

After setting up the necessary parameters in the DOS system and copying the files from the distribution disk to their complimentary directories on the hard disk you are ready to compile SPH.

First, make the changes in the file SPH.H (found in the directory \sph\code\sphlib) as specified in the introductory paragraph of this section. Next, go to the directory \sph\code\curlib and type MAKE. This will execute the batch file MAKE.BAT and will create the CURSES library CURLIB.LIB. Go then to the directory \sph\code\plib and again type MAKE. PLIB.LIB will then be created. Now go up to the directory \sph\code. Again type MAKE and the SPHLIB routines will be compiled and put in the library called SPHLIB.LIB located in the subdirectory \sphlib. This batch routine will also compile SPHC.C, the SPHC driver, and SPH_INIT.C, the input problem code. All the afore mentioned libraries will be linked with the SPHC and SPH_INIT object codes to produce the executable SPHC.EXE. To execute SPHC simply type SPHC and the input menus will appear. At this point you may change any of the input parameters interactively before problem execution begins.

If you wish to run a *different problem*, get the desired input code from \sph\tests and rename it to SPH_INIT.C in the directory \sph\code then type MAKE SPH_INIT. Just SPH_INIT.C will be compiled and the object files and libraries will be relinked. If you wish to change a physics routine, edit your changes and type, while in the directory \sph\code, MAKE routine_name. This will compile and install the new routine into SPHLIB.LIB and relink to create a new executable.

TECHNICAL GUIDE CONTENTS

3.0 TECHNICAL GUIDE

3.1 INTRODUCTION

This is a summary of the numerical and physical approximations and algorithms currently implemented in the SPHC code at AMRC. As the code is developed, this document will change accordingly. Further details on many of these techniques, as well as alternate approaches, can be found in the SPH literature.

3.2 VARIABLE SMOOTHING LENGTH

For problems involving density changes of more than a factor of 3, a variable smoothing length must be used. The intention of such a scheme is to vary the size of individual particles in much the same way as a Lagrangian grid. Two approaches have been tried to date:

1. Nearest neighbor: here at the beginning of the hydro step, the smoothing length of each particle is set to the distance between the particle and its nearest neighbor. This is a foolproof method, but has the following problems: 1) either the new h is computed in an additional N^2 loop at the beginning of each step, or it is computed during the previous step and needs to be saved for the next step, thus entailing extra time and/or storage, 2) particles may pair up, with no further interactions, 3) the repulsive force between adjacent particles may be too big, causing oscillations.

2. Density scaling: here the smoothing length is scaled directly to the density change raised to the power (-1/dimension), resulting in a smoothed version of the above algorithm. The problem here is that the method may be too smooth, allowing particles to "stack" at a point with no repulsive tendency, and, as observed in one instance, there is no reason why a single particle cannot go to the zero-size/infinite density limit. This behavior can probably be avoided be careful choice of initial conditions.

   A third approach has been implemented by W. Benz:

3. We write the time derivative of h: dh/dt in terms of d(rho)/dt:

$$dh/dt = (-1/d)\ (h\ /\ rho)\ d(rho)/dt$$

and use:

d(rho)/dt = -rho Div(v).

Here d is the dimension of the calculation, and v is the particle velocity. This allows us to integrate h as an additional equation in the Runge-Kutta computation. The advantage here is that h now is updated in step with the other variables, and the Runge-Kutta error control keeps dh/dt in check automatically. This produces a more stable variation, avoids overshooting in the density, and performs better in general than the other approaches.

The current algorithm in SPHC consists of three steps: 1) upon problem initialization, the user sets the smoothing length of each particle in sph_init(), 2) using this guess to find neighbors, on the first step of a problem the smoothing length is reset to the nearest-neighbor value, and 3) during the computation, the smoothing length of each particle is scaled by the density.

We intend to try switching to the Benz scheme in the near future.

3.3 PERFECT GAS EQUATION OF STATE

When part[i].mat_no = 0 for any particle, the perfect gas equation of state is selected. This option activates the use of three global parameters:

Gamma = Cp / Cv = the ratio of specific heats,
GammaP = the polytropic index (defined below), and
Mu = the mean molecular weight (A / (Z+1) for the fully ionized case).

The equations used in eos() are:

P = (R / Mu) * rho * T, and
T = (R / Mu) * (Gamma - 1) * E,

where R is the gas constant. The inverse of the first equation is used to generate the initial energies from specified temperatures.

The polytropic gamma is used as follows: we wish to specify a constant overall specific heat:

dQ/dT = C = constant.

so

dE/dt = dQ/dt - P dV/dt

becomes

dE/dt = (C/Cv) dE/dt - P dV/dt.

If we define

GammaP = (Cp - C) / (Cv - C),

then the energy equation can be written

dE/dt = ((1 - GammaP) / (1 - Gamma)) (-P * dV/dt).

and in the perfect gas case, you can easily show

P = K1 * rho $^{\text{GammaP},}$ T = K2 * rho $^{(\text{GammaP}-1),}$ etc.

This is a generalization of the usual adiabatic perfect gas treatment. See Chandrasekhar: "Stellar Structure", Chapter II, Section 3 for a detailed discussion of this approximation. Useful special cases are:

C = 0 = > GammaP = Gamma = > adiabatic case as usual,
C = infinity = > GammaP = 1 = > isothermal case, all Gamma ! = 1,
C = Cp = > GammaP = 0 = > isobaric case,
C = Cv = > GammaP = infinity = > constant sp. volume (density).

The polytropic correction to dE/dt is performed in poly(), located in eos.c.

3.4 ANALYTIC EQUATION OF STATE

This section describes the analytic equation of state developed for laser/plasma modeling. The intention is to obtain reasonably accurate estimates of the pressure, temperature, and composition as functions of density and internal energy for a wide range of energies and densities. The inverse function (energy as a function of temperature and density) is also required for problem initialization. See the User_guide for a list of eos parameter values.

For the following discussion we assume that the temperature and density are given, since this simplifies the formulae.

<u>The Cold Curves: T = 0</u>

The pressure and internal energy per gram at zero temperature are taken to be the Grueneisen values (see Harlow and Amsden, LA 4700, p. 3):

$$P_0 = c0^2\, dV / [V0 - s dV]^2 \quad , \qquad (2)$$

$$E_0 = P_0\, dV/2 \quad ,$$

where

$$dV = V0 - V \quad ,$$

$$V0 = 1/\text{solid density} \quad ,$$

$$s = (\gamma_G * (V/V0) + 1)/2 \quad , \qquad (3)$$

$$\gamma_G = \text{Grueneisen gamma} \quad ,$$

$$c0 = \begin{array}{l}\text{solid sound speed if } V <= V0 \\ \text{liquid sound speed otherwise}\end{array} \quad ,$$

The values for c0 and V0 are taken from the literature, and that of γ_G is usually found in the Los Alamos Sesame Handbook, LA=10160-MS. γ_G varies inversely with density. The value of γ_G in these tables and the Sesame handbook is that at V = V0. The (V/V0) multiplier to γ_G eliminates the singularity at V = V0 (s - 1) / s.

Note that P_0 and E_0 are exactly zero at V = V_0. These analytic expressions for E_0 and P_0 diverge significantly from the Sesame handbook representations of the cold curve for values of V/V0 < 0.5.

The Pressure

In all cases, the pressure is taken to be the sum of the cold curve and a perfect gas contribution:

$$P = P_0' + (R/\mu)\,\rho\,T. \tag{4}$$

Here R is the gas constant = 8.317e7 erg/deg/mole. The value taken for the mean molecular weight, μ, will be discussed below. P_0' is given by

$$\begin{aligned} P_0' &= P_0\,(1 - T/T_{cp})\,, \\ P_0' &= 0 \qquad T > T_{cp}, \end{aligned} \tag{5}$$

where T_{cp} is the critical point temperature. This ensures perfect gas behavior at temperatures above the critical point.

At low temperatures and densities the pressure is negative (van der Waals attraction). In some circumstances the magnitude of this negative pressure can be large, so for our purposes the pressure is always limited to be larger than -1.e9 dyne/cm^2.

Solid / Liquid State: $T < T_{vap}$

Here μ = the molecular atomic weight, $\gamma = \gamma_{liq}$, and the energy is given by:

$$E = E_0 + (R/\mu)T/(\gamma - 1). \tag{6}$$

The value for γ_{liq} is chosen to fit the SESAME energy variation in this region. It can be near unity to simulate the various excitations, dissociation, and even ionization possible in these temperature ranges. T_{vap} is usually chosen to correspond to the temperature at which negative pressure disappears in the Grueneisen formula discussed above. This temperature is somewhere between the STP vaporization temperature and the critical point temperature, as appropriate for a pressure-driven phase transition.

Molecular Gas: $T_{vc} < T < T_{diss}$

Here, as above, μ = the molecular atomic weight, $\gamma = \gamma_{mol}$, and the energy is given by:

$$E = E_0 + (R/\mu)T/(\gamma - 1) + E_{vap}. \quad (7)$$

The vaporization energy, E_{vap}, is taken to be the sum of the handbook values of the melting and vaporization latent heats.

Atomic Gas: $T_{diss} < T < T_{low}$

Here μ = the mean atomic weight, $\gamma = 1.6667$, and the energy is:

$$E = E_0 + (R/\mu)T/(\gamma - 1) + E_{vap} + E_{diss}. \quad (8)$$

The dissociation energy, E_{diss}, is the handbook value. Note that the second term has changed in value due to the switch in γ. This means that the various excitation energies in the complex molecular state have all contributed to the dissociation process and are no longer present in the atomic gas.

Ionization: $T_{low} < T < T_{high}$

Here $\gamma = 1.6667$, μ varies with composition.

The intention here is to eventually implement a non-LTE set of rate equations to handle ionization in the general case. For now, a very simplified model has been adopted. The ionization energy is assumed to scale with the square of the mean atomic charge (or can be specified), and increases linearly between T_{low} and T_{high}, taken as 1 eV and 2 eV * Z^{24}. The mean molecular weight is chosen in such a way as to make the total energy vary linearly with temperature as well. This facilitates inversion of the equation and results in a physically reasonable rapid variation of μ at initial ionization, slower variation at higher levels of ionization. Here

$$E = E_0 + (R/\mu)T/(\gamma - 1) + E_{vap} + E_{diss} + E_{ion}(T). \quad (9)$$

Note that an estimate of the number of free electrons per ion may be obtained from the routine nfel(T, Z) for physics applications.

Fully Ionized Plasma: $T_{high} < T$

Here $\gamma = 1.6667$, μ = (mean atomic wt) / (mean atomic charge + 1), and the energy is:

$$E = E_0 + (R/\mu)T/(\gamma - 1) + E_{vap} + E_{diss} + E_{ion-tot}. \tag{10}$$

Adjustable Parameters

In this formulation the only free parameters are γ_{liq} and $c0_{liq}$, whose values are chosen by fitting to the low temperature region of the Los Alamos SESAME tables. The following values are usually known, but may also be adjusted slightly to improve the fit to the tables: T_{vap}, T_{diss}, T_{low}, T_{high}, and γ_{mol}.

Limitations

The cold curves used here are not applicable for pressures of over 1 Megabar in solid material shocks. The ionization model is at present very crude, and does not include any effects of density variation. The details of the solid/liquid/vapor phase transitions are not treated. "Glitches" seen in the tables in this region do not appear in this formulation, except at T_{vap}. Also the density dependence of the various transition temperatures is ignored here.

3.5 PROBLEM GEOMETRY

The normal mode of computation on SPH is 3-dimensional Cartesian coordinates. One- and two-dimensional runs are permitted as well by setting the global parameter "dimension" = 1, 2 or 3. This affects the normalization of the kernel (see the routine kernel.c), certain diagnostics, and the size of the restart dumps, but the computation is always carried out in a full three-dimensional vector formulation.

In all cases, the normalization of the kernel is chosen such that its integral over all space is unity.

Two other options are included for use in lower dimensionalities:

Spherical; on.sphere = TRUE:

For 1 dimensional problems only. The x coordinate becomes the radial "r". The kernel is further normalized by $(4 \pi x^2)$ to simulate a spherical shell of radius x, rather than an infinite slab. The origin is at $x = 0.$ and a wall boundary should be placed at that location.

The renormalization is not done in the kernel itself, but in the calling routine. The reason for this is that the renormalization must correspond to the particle mass that multiplies the kernel, in order to refer to the correct particle.

Note, in problem setup to achieve a constant density with uniform zoning the radial weighing factor will need to be applied to the particle masses.

In this case if the smoothing length is variable the radial weight is also applied here to remove the effect of geometry.

Note that the gradient is not affected by this transformation, but occurrences of the divergence and curl will need to be modified.

Cylindrical; on.cylinder = TRUE:

For 1- or 2-dimensional problems. The modifications are identical to the spherical case, except that the weighing factor is $(2 \pi x)$, and the x coordinate becomes the radial "rho", while the y coordinate represents the cylindrical "z". Thus the axis is always taken to be in the Y direction.

In the 1-dimensional case, we imagine a cylindrical problem that depends only on the radial coordinate (cylinder seen end on). In the 2-dimensional case, the solution can depend on the radial coordinate and on the coordinate along the axis symmetry (taken to be "y" in this code, usually referred to as "z"). In both cases the solution is independent of the theta coordinate.

3.6 LASER DEPOSITION

The current version of SPH includes a laser deposition algorithm that is efficient and quite accurate in most cases. In most laser-target models of interest the deposition of energy is due to inverse Bremsstrahlung in the underdense material, together with resonant processes that occur near the critical density. For the laser/target/ambient problem, the physics can be divided into two regimes: 1) the underdense ambient material, possibly pre-ionized, and 2) the hot target material, where the bulk of the deposition occurs near critical. In the first case, it is sufficient to ignore absorbed laser energy and compute the deposition directly for each particle using the full area-weighted laser energy as input. For the second, more care is needed, but we primarily need an algorithm to identify the particles near critical density and distribute the laser energy properly.

The present approximate scheme is implemented in the routine laser_dep(), and works as follows: First, a density, usually 1/2 critical, is chosen as the "deposition density". On each timestep, a linked-list is made of all particles that fall within the laser beam. A "shadow-radius" (taken to be the smoothing length) is chosen for each particle, and particles that are shadowed by "overdense" (density greater than the dep-density) particles are removed from the list. Shadows are shaped as follows:

```
                  ******
                **      **
   ^           *    /\    *
   |           *   /  \   *
   |            **/    \**
   X             /******\
                 |      |
                 |      |
                 | shad-|
                 |  ow  |
```

The shadows are properly shaped to converge on the laser focal point (see below). The current algorithm involves selecting the particles in the laser beam, sorting them on distance from the focal point, and removing the shadowed particles working toward the laser. This operation is now written in vector form to allow changes in the dimension, geometry, and laser f-number.

Of the remaining particles, the underdense are then heated according to the inverse Bremsstrahlung formula (see AMRC-R-1031, pg. 45, Eq. 16), assuming small total absorption in this region (0.1 percent is expected). The remaining laser energy is deposited in the unshadowed overdense particles with uniform energy/mass (weighted by the function Spot_prof()).

SPH allows the f-number of the laser to be specified, defined as:

f = focal length / aperture

therefore, the laser beam half-angle of convergence is:

theta = arcsin (1 / (2f)).

This is implemented by increasing the spot area by the appropriate factor in both the particle-list code and the deposition code. This algorithm should work behind the target and beyond the focal point as well. Taking the f-number to be very small results in illumination over a complete sphere.

The deposition is computed via the fraction of laser energy absorbed. The routine laser_pulse() returns the laser power in ergs/cm^2/sec, laser_energy is the energy delivered in the current timestep, und_energy is the energy deposited in the underdense region, and crit_energy is the remaining energy. For the 2D and 3D cases, the energy deposited in a particle scales as the fraction of spot area subtended in the underdense region.

3.7 NOH SHOCK FOLLOWING TECHNIQUE

This option is needed to prevent spurious heating in problems involving hypersonic collisions of different regions, or a region and a wall boundary. It is invoked by setting the artificial viscosity parameters g1 and g2 in the "Control Parameters" menu (generated in eos_init()). The normal values for these parameters are:

normal problems - g1 = g2 = 0,

problems with collisions - g1 = 0.5, g2 = 1.0 .

These parameters turn on linear and quadratic artificial thermal diffusion terms in the viscosity treatment. If either of the parameters are non-zero, the flag on.av_diff is set to TRUE in eos_init(), the default value is FALSE. The computation of the diffusion coefficient is done in hydro(), using the quantity rhodelv[] computed in rho_compute(). The coefficient is added to part[i].thm_con for use by the routine thm_diff().

For complete details on the technique, see Monaghan: "SPH and the Shocks of Noh", (preprint, 1988).

3.8 ELECTRON THERMAL CONDUCTION

A simple explicit implementation of heat conduction via electron thermal diffusion including a flux limiter has been added. The option is controlled by the flag on.elect_diff, set in setup(), menu 1.

The computation of the coefficient is done in e_diff() using the formula

$$t_c = 3.58e - 5 * \left(nfe/Z^2\right) T^{2.5}, \tag{11}$$

where nfe is the number of free electron/ion (gotten from nfel()), Z is the atomic charge, and T is the temperature. The coefficient is added to part[i].thm_con for use by the routine thm_diff(). Since the thermal flux is not available in the current diffusion package, the flux limit is crude and probably overly conservative. This will be fixed in a future upgrade.

3.9 BOUNDARIES

The boundaries in the present implementation of SPHC are controlled by several structures, all specified in sph.h, verify the current version for changes. Type and number are given by the following declarations:

```
/* boundaries */
#define MAX_BOUNDARIES 6

/* boundary types  */
#define WALL   1
#define PISTON 2
#define PER    3
#define REFL   4
#define FLOW   5
#define AXIS   6
#define FIXED  7

/* Periodic types - several can be on at once */

#define  X2  1
#define  Y2  2
#define  Z2  4
```

The parameters used during the computation are controlled by the following structures:

```
struct bdry_list {
    struct vector coord;
    int num;
    int crossed; /* TRUE - particle is on the wrong side of boundary */
                 /* will be returned to problem space according to boundary
                    conditions by check_bounds */
};

struct bound {
    int type;
    /* for Periodic only */
    /* Periodic boundaries must be two parallel boundaries in X,Y, or Z
       No checking is performed to verify consistency,
       the offset is subtracted from the appropriate boundary */
    int per_type; /* 2X, 2Y, 2Z */
    struct vector per_offset;
};
```

The logic controlling the boundary computation is given by the detailed flow chart in the SPH Programmer's Guide.

The details of the WALL boundary have been discussed by Campbell in the report MRC/ABQ-R-1100. In this case we restore the boundary terms usually dropped in the integration by parts. This results in a boundary acceleration that is given by

$$dv_i/dt = 2\sum_j \left(P_j/\rho_j^2\right)\, W_{ij}\, r_{ij} \;/\; |r_{ij}|^2. \tag{12}$$

Where W_{ij} is the kernel (note: not the gradient), and $r_{ij} = r_i - r_j$. In addition the boundary contributes to the density of a nearby particle as if a mirror particle were located at the boundary. A particle approaching the boundary also feels a viscous interaction, needed to handle shock reflections. The boundary force is always normal to the boundary.

The PER, AXIS and REFL boundaries are handled using ghost particles. In the case of periodic boundaries, a particle near the wall interacts with particles in its own vicinity, as well as those near the opposite boundary. This is accomplished at present by copying each of the ghost particles into particle location "0" as needed, so additional memory for the ghosts is not needed, but some computational overhead is involved. The reflecting boundary is similar, except that the particle sees a reflection of all nearby particles across the boundary. At present, the AXIS boundary is taken to be WALL.

The planned PISTON and FIXED boundaries will have specific parameters attached to the boundary, which may be moving. The planned FLOW boundary will allow particles to travel through the boundary without further acceleration or heating, to be dropped from the problem beyond 2h.

3.10 PARTICLE DIVISION AND COMBINATION

SPH is a Lagrangian technique. This is advantageous for problems with small density gradients, or those in which only the high mass regions are of interest. In some cases other types of problems can be handled in a Lagrangian scheme by varying the particle mass in different parts of the problem. This can lead to problems, however, if mixing causes the different mass particles to change position. In general, a Lagrangian code needs some means to divide and combine cells dynamically during the computation. This note describes such a scheme for SPH.

There are two situations requiring the addition or subtraction of particles from an SPH run in progress. The first is that of a problem with inflow or outflow boundaries. This is primarily a bookkeeping problem. If a particle is to be added to the list, the particle count in incremented and the new particle's characteristics copied into the new structure location. If a particle is to be deleted, the last particle in the list is copied over the structure location of the particle to be dropped, and the particle count decremented. In both cases care needs to be taken to ensure that adjacent particles are not strongly affected by the change.

The second type of problem is one in which the local density changes are large, and spatial resolution needs to be maintained at a constant level. A rarefaction wave is an example of this type, where the density decreases exponentially in time and space. As the density decreases, the particles move apart, and if the smoothing length is held constant, the particles soon separate and the computation is at an end. If the smoothing length is allowed to vary, the computation can proceed, but resolution becomes increasingly poor at low densities. In order to maintain the needed resolution at low densities, the particle size at the initial density must be made inordinately small, which decreases the Courant time and greatly increases the expense of the computation.

3.10.1 Division:

A solution to this problem is the following particle divide algorithm, which depends on the dimension of the computation. In all cases, a threshold is specified for the largest allowable particle size. When this threshold is reached by a particle, it is divided as follows:

1D: Here the particle is divided in half, each offspring having half the mass and half the smoothing length of the original. The two offspring are placed with a spacing equal to their new smoothing length, centered on the old particle position.

2D: Here the particle is divided into three equal portions, each with 1/3 the mass and smoothing length equal to 1/sqrt(3) = 0.577 of the original. They are placed at the vertices of an equilateral triangle with sides equal to the new smoothing length, centered on the old particle position, and with a random orientation.

3D: Here the particle is divided into four equal portions, each with 1/4 the mass and smoothing length equal to 1/cbrt(4) = 0.63 of the original. They are placed at the vertices of a tetrahedron with sides equal to the new smoothing length, centered on the old particle location, and with a random orientation.

In all cases the velocity of the new particles are chosen to be equal to the velocity of the parent particle, thus exactly conserving mass, momentum and energy. This prescription replaces the parent particle with a set of offspring that approximate the original in size and shape, and thus cause minimum disruption to neighboring particles.

3.10.2 Combination:

Recombining particles is a more difficult operation, because of the greater likelihood of disruption of the immediate vicinity. In two identifiable cases, however, the recombination is easily done:

1. If two particles somehow approach each other much more closely than a smoothing length, and share the same velocity, they may never separate if a flat-topped kernel is being used. In this case the two particles may safely be combined into a single particle with the sum of the masses and a velocity equal to that of the original particles.

2. If a small mass particle finds itself near one of much larger mass (within a smoothing length, say), then it may safely be absorbed by the larger particle with little effect on the computation. This type of combination is probably needed to avoid numerical difficulties in mixed regions.

In the general case of recombination whenever the resolution is too great in a given region a more subtle scheme is needed. One approach would be a redefinition of all the particle masses, positions, and velocities in a region of space. Another approach might be to combine the particles in groups of two or three that satisfy a criterion for least disruption of neighbors. Finally, it may be possible to allow some flexibility in the recombination and iterate until a smooth, conservative solution is obtained. None of these schemes have been tested at present.

3.10.3 Examples:

Figure 1 shows the result of a division operation in an expanding fluid. Notice that the "before" and "after" shaded areas are nearly equal. The densities of the three particles after the division are slightly lower than that of the original, but this depends on the surrounding particles as well.

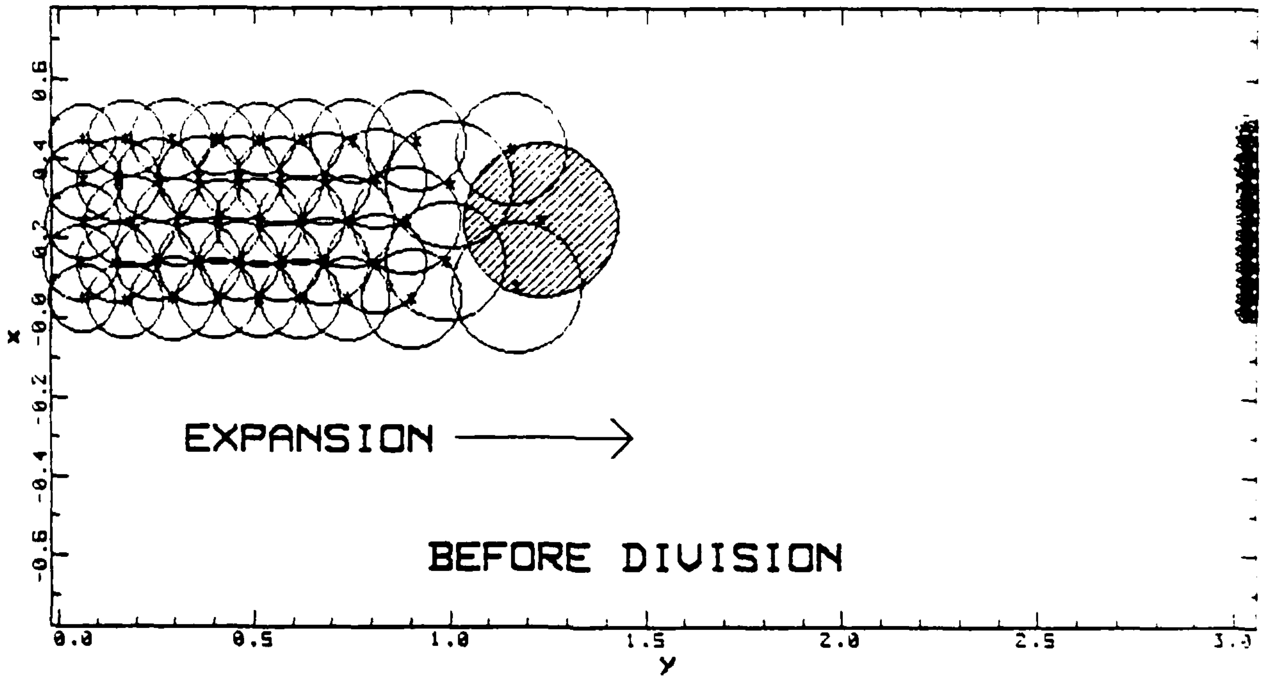
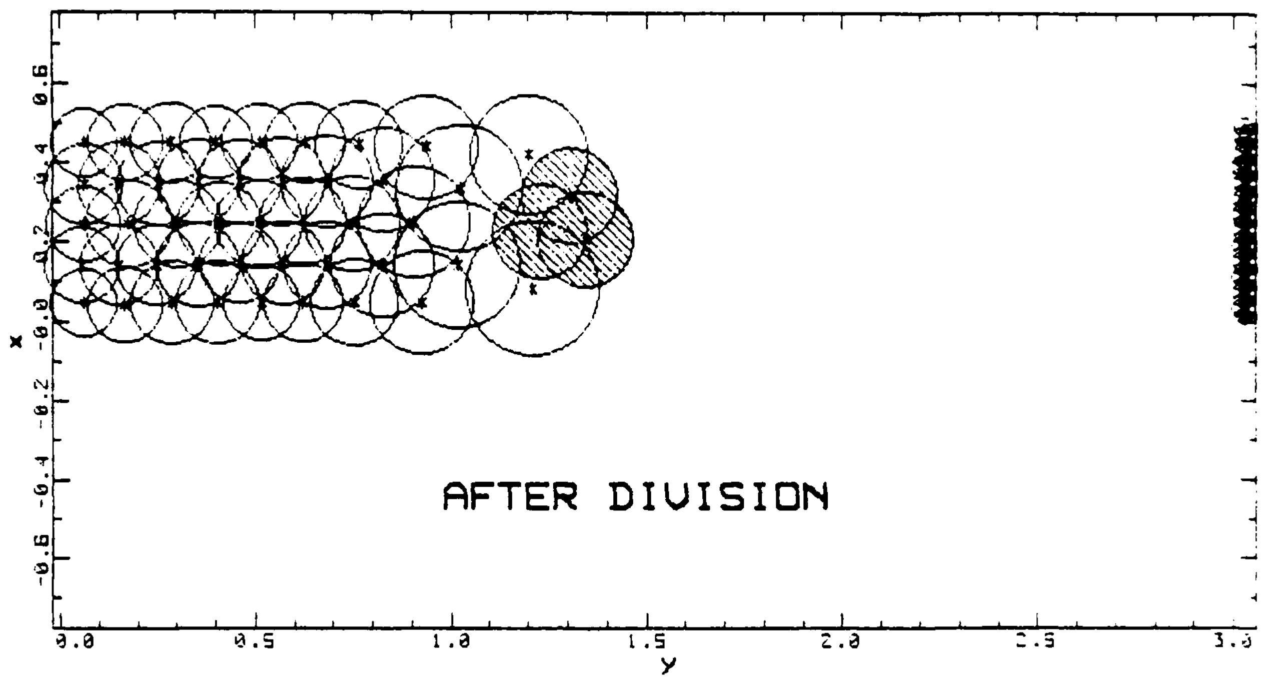


Figure 1.

R-1237

Figure 2 shows the subsequent evolution of this test, in which an expanding gas accelerates a high-density foil through a short tube. The computation starts with 82 particles, and ends with 450. Adequate resolution is maintained at all stages of the computation in spite of extreme low densities in the rarefaction regions. No recombination was used in this computation, and a minimum mass of 1/10 the original was specified.



R-1237

Figure 2.

3.10.4 Discussion:

This discussion demonstrates that particle division is possible and useful in SPH. The algorithm presented here is rather crude, and refinements can be added to suit the problem. Here the division is triggered by a lowered density. A more sophisticated algorithm, easily implemented, would divide particles in regions of increased gradients. This would allow a detailed tracking of shocks, for example. Obviously, more complicated schemes can be concocted, based perhaps on the currently used ALE adaptive grid algorithms.

FUNCTION DIRECTORY CONTENTS

4.0 FUNCTION DIRECTORY

4.1 INTRODUCTION

This is an alphabetical list of all the major functions in the SPHC package. See the Programmer's guide for a standard template to use for each entry. In most cases, the listing is by routine name, but in some cases (the vector routines, for example), entire modules are included as a single entry.

4.2 ALPHABETICAL LIST OF ALL SPHC FUNCTIONS

```
BURN MODULE:  burn.o
--------------------
    functions:      burn_yield()
                    burn_part()
                    sigmav_bar()

    type:    void

    arguments:  for burn_part()
        mat.no      material number, only 6 (DT) and 10 (DD) are valid
        rho         particle density (gm/cc)
        mass        particle mass (gm)
        temp        particle temperature (Kelvin)
        dt          time step for this cycle (sec)
        neutrons    number of neutrons produced by this particle
                    during this cycle (returned)
                    for sigmav_bar()
        mat         particle material (DD or DT only)
        temp        particle temperature (KeV)
        svbar       particle <cross-section * velocity> (cm**3/sec)
                    (returned)

    calls:  each other

    principal global variables:
        on.burn         burn flag
        total_neutrons  sum of neutrons over all particles over time
```

description:

This is the neutron yield module for DD and DT reactions (burn). Burn_yield() is simply a loop over all particles each time step to sum their individual contributions. Burn_part() calculates the number of neutrons produced by an individual particle. Sigmav_bar() calculates <cross-section * velocity> for DD and DT reactions at the particle temperature.

references:

Sigmav_bar() is an analytic fit to the cross-section curves found in S. Glasstone and R. H. Loveberg, Controlled Thermonuclear Reactions, Van Nostrand Reinhold, 1960, page 19.

```
check_boundaries()
------------------

 module:              sph_bounds.o

 type :               int

 arguments: none

 calls: sph_bc()
```

description:

Checks each particle to see if any have left the problem space and moves (by reflection or periodicizing?) them back into problem space.

```
copy_particle()
---------------

    module:          sph_bounds()

    type:            int

    arguments:
       i             source particle number
       j             destination particle number
```

description:

Copy a particle to another slot.

```
delete_particle()
-----------------

    module:         sph_utl()

    type:           void

    arguments:      i    particle number

    calls:
                    copy_particle()
```

description:

This routine copies particle Npart over location i and decrements Npart, thus dropping particle i from the computation.

```
divide_particle()
-----------------

    module:         sph_utl()

    type:           void

    arguments:      i    particle number

    calls:
                    copy_particle()
                    vector routines
                    write_part()
```

description:

This routine will substitute for particle i a number of offspring depending on dimension (1D-2, 2D-3, 3D-4), and update Npart accordingly.

```
divide_particles()
----------------

    module:         sph_utl()

    type:           void

    arguments:      none

    calls:
                    divide_particle()
```

description:

This routine loops over all of the particles, tests if h > max_h (smoothing length), and if so calls divide_particle().

```
do_bounds()
----------
 module:             sph_bounds.o

 type :              int

 arguments:  part_num - a particle to interact with the boundaries.
             do_wall() - a function to compute wall-particle interactions.
             do_ghost() - a function to compute particle-particle
                 interactions using ghost particles.

 calls:
         sph_bc() - user routine to check for boundaries
         do_refl() - which calls do_ghost()
         do_per() - which calls do_ghost()
         do_wall()
         get_refl_list()
         get_per_list()
```

```
do_hydro()
----------
 module:              hydro.o

 type :               void

 arguments:
     i, j             particle numbers for interaction

 calls:
                      vect3 routines
                      kernel()
                      art_visc()

 principal local variables:
     termi,j     P/rho^2 for particle i, j
     av_ij       artificial viscosity ij
     termij      termi + termj + av_ij
     multi,j     additive term to acceleration of particle i, j
     term1,2     linear and quadratic terms in Noh conductivity
```

description:

This routine does the hydrodynamic contributions to accel, dedt, and thm_con. Note that the renormalization of the kernel for spherical and cylindrical coordinates are done here to allow the proper adjustments to be done for the ith and jth particles.

references:

See Monaghan: "Shocks of Noh" for the conductivity term.

```
do_per()
--------

 module:              sph_bounds.o

 type :               int
```

```
arguments:  i - a particle number.
            part_list - list of neighbor particles
            bdry_coord - the point nearest part_coord on the boundary in
                question
            do_ghost() - particle-particle function

calls:              copy_particle()
                    do_ghost()
```

description:

Wraps each particle on part_list around the periodic boundary and interacts particle i with the reflections using do_ghost().

```
do_refl()
----------

module:             sph_bounds.o

type:               int

arguments:  i - a particle number.
            part_list - list of neighbor particles
            bdry_coord - the point nearest part_coord on the boundary in
                question
            do_ghost() - particle-particle function

calls:  reflect_particle()
        copy_particle()
        do_ghost()
```

description:

Reflects each particle on part_list and interacts particle i with the reflections using do_ghost().

```
do_rho
------
    module:         rho.o

    type:           void

    arguments:
        i,j         particle numbers to compute

    calls:
                    vec3 routines
                    kernel()
```

description:

Compute the density contribution of particle i to particle j and vice-versa. Also compute the ij the term of rhodelv.

```
drop_particles()
-----------------

    module:         sph_utl()

    type:           void

    arguments:      none

    calls:
                    sph_bc()
                    delete_particle()
```

description:

This routine checks all Npart particles for deletion by calling sph_bc() and looking for a return value of -1. If obtained, the particle is deleted from the computation.

```
dt_compute()
------------
    module:        SPHCompute.o

    type:          void

    arguments:     none

    calls:         vect3 routines

    principal global variables:
        dt_phys     "physics (Courant) timestep"
        dtflow      flow timestep structure
        dthyd       hydro timestep structure
        dtgrav      gravity timestep structure
        dtthm       thermal diffusion timestep structure
```

description:

This is the timestep computation. First the various "physics" timesteps are computed as required by an explicit integration. Introduction on new physics in the code requires modification of this section. The minimum of these is the "physics timestep" dt_phys. The screen summary is then printed. Now the actual timestep is computed using the Runge-Kutta error criteria, as well as limits based on the physics timestep. If a retry is indicated, the variable and parameter resets are done here. All the logic for special cases, such as relaxation and no dt control are done here as well.

```
e_compute()
-----------
    module:        SPHCompute.o

    type:          void

    arguments:     none

    calls:         vect3 routines

    principal global variables:
        k_e         kinetic energy
```

t_e	thermal energy
p_e	potential energy
en0	initial total energy
en_tot	current total energy

principal local variables:

e0_tot	initial energy including sources
check	energy check (%)

description:

Here we compute the total energies for the current configuration, and perform the energy check by direct comparison with the initial energy. The result is printed to the screen.

```
ediff()
-------
```

module:	sph_utl.o
type:	void
arguments:	none
calls:	nfel()

principal global variables:

Thm_mult	thermal conductivity multiplier (default=1)
Flux_lim	flux limit parameter (default = 0.2)

principal local variables:

t	temperature (K)
mat	material number
nfe	number of free electrons/ion
ne	number of free electrons/cc
z	atomic charge
a	atomic number
tc	thermal conductivity
f_lim	thermal flux limit

description:

Compute the thermal electron conductivity

references:

Spitzer, Cox and Guili "Principles of Stellar Structure", Sect. 16.123.

```
EOS MODULE: eos.o
-----------------
    functions:      eos_init()
                    eos_pe()
                    nrg()
                    eos_pt()
                    t_nrg()
                    mat_lookup()
                    art_visc()
                    poly()

    type:           void

    arguments:
        mat         input variables:  material, temperature (K),
        temp        density (g/cc), and specific internal energy (erg/cc)
        rho
        energy

        press       pointers to returned variables:  pressure (dyne/cm2),
        ener        energy (erg/cc), sound speed squared (cm/s)^2, and
        csq         debug flag
        dbg

    calls:          each other

    principal global variables:
        Mu          these used for perfect gas case only:  mean mol.
        Gamma       weight, ratio of specific heats, polytropic gamma
        GammaP
        eos tables:  see Tech_guide notes on eos
```

description:

This is the equation of state package. In its present form, it is designed to be a simple semi-realistic treatment over a wide range of parameters. At solid/liquid densities it uses a Grueniesen analytic formulation, which is faded out until at the triple point the form is fully perfect gas. The overall equation uses 15 parameters, most of which are physical constants of the material, but some of which are adjusted to obtain a better fit to the LANL SESAME tables.

A separate, but related routine is nfel().

references:

See the comments at the top of eos.c, and the sections in the User Guide and the Technical Guide for details.

```
eos_compute()
-------------
    module:         SPHCompute.o

    type:           void

    arguments:      none

    calls:          eos_pe()
                    eos_pt()
```

description:

This is the equation of state front-end routine. It is expected to undertake all actions necessary to complete the computation of specific energy, pressure, temperature, and density. Presently, it calls either the "pe" or the "pt" eos lookup, does some simple data checks, and loads the sound speed.

```
finish()
--------
    module:          run_check.o

    type:            void

    arguments:
        ierr         exit code, 0 = normal, 1 = other

    calls:           write_step()
```

description:

This is the official exit i/o routine. It writes a dump to err_dump if ierr != 0, prints a summary of run information to the screen, and returns to the calling program.

```
get_nbr_list()
--------------
    module:          "tree" (see NEIGHBOR MODULES)

    type:            structure particle *

    arguments:
        i            particle #, (if reversible), 0 (otherwise)
        loc          pointer to location vector
        h            smoothing length

    calls:           neighbor entry routine

    principal local variables:
        list         pointer to last particle in the list
```

description:

This routine returns a list of particle neighbors, given the location and size of any particular particle. In addition, provision is made for "reversible" processes (conservative processes requiring only the above-diagonal entries in the particle-particle matrix) by specifying a particle number. If this parameter is zero, all neighbors are returned. The particle itself is always included in the list.

```
get_per_list()
--------------

 module:               sph_bounds.o

 type :                int

 arguments: i - a particle number.
             part_coord - the coordinate of a point in problem space
             sm_ln - smoothing length around coordinate
             bdry_coord - the point nearest part_coord on the boundary in
                 question

 calls:  get_nbr_list()
```

description:

Wraps part_coord to matching boundary and uses get_nbr_list to find the neighbors to reflect.

```
get_refl_list()
---------------

 module:               sph_bounds.o

 type :                int

 arguments:  i - a particle number.
             part_coord - the coordinate of a point in problem space
             sm_ln - smoothing length around coordinate
             bdry_coord - the point nearest part_coord on the boundary
                 in question

 calls:                get_nbr_list()
```

description:

Reflects part_coord across the boundary and uses get_nbr_list to find the neighbors to reflect.

```
gravity()
---------
    module:         hydro.o

    type:           void

    arguments:      none

    calls:          vlin()

    principal global variables:
        on.grav     set if gravity is on
        grav        vector specifying the constant acceleration
```

description:

Apply a constant gravity acceleration to all particles. The magnitude of the acceleration is set in the input decks or menus.

```
grid_gen()
----------
    module:         grid_gen.o

    type:           void

    arguments:
        part_init   pointer to a function that initializes particles
                    located in the sph_init.o module

        dir         = 0...align hex grid along x axis
                    = 1...align hex grid along y axis


    calls:          "part_init"  routine at each grid point
```

description:

Grid generator. User requests a certain type of grid covering a specified volume of space, this routine will determine the spacing and particle mass required and call the "part_init" routine once for each grid point to allow the user to initialize the problem in a completely general way. Details:

```
in 1D:  generates a uniform spaced grid
in 2D:  generates a hexagonal grid, spacing = h_spacing
in 3D:  generates hex grids in all cartesian directions
        note:  this is not a close-packing solution

in sph_init0 must set the following quantities:
    Grid_gen = TRUE
    Grid_standoff = amount to stand off from boundary, units of
        spacing
    Space_adjst:  use to adjust particle spacing (multiplies it)

in sph_init do the following
    Grid_type = 1...rectangular,  = 2...hexagonal
    Rho:  density of material
    Volume:  volume of current grid
    Grid_npart = number of particles desired in current grid
    Maxx, Minx, Maxy, Miny: define a cube containing the current grid

    call grid_gen( "part_init" )  for first grid
        if return == 0, abort
        if return == 2, mem exhausted
    reset any parameters for another grid
    call grid_gen( "part_init.2" )  for second grid

    repeat for additional blocks, reset all changed parameters

in "part_init" do the following things:
     1.  test Grid.x, Grid.y, Grid.z, if in desired volume, if not
         return(1);
    2.  else increment:  Part_num++
        initialize part coord = Grid.x, Grid.y, Grid.z,  mass = Mass
        initialize part veloc, temp, mat_no
    3.  return(1)
    4.  if any fatal errors occur, return(0)
```

```
h_compute()
-----------
    module:          SPHCompute.o

    type:            void

    arguments:       none

    calls:           get_nbr_list()
                     vect3 routines
```

description:

This routine performs an explicit update of the smoothing length. Note that an estimate of the smoothing length must be set in the sph_init() routines to determine the neighbors for the first pass. A special algorithm is used the first time through: the distance to the nearest neighbor is taken to be the smoothing length of the particle. Subsequently, the smoothing length is scaled with the -1/nth root of the density, where n is the dimension of the problem.

```
hydro()
-------
    module:          hydro.o

    type:            void

    arguments:       none

    calls:           get_nbr_list()
                     do_hydro()
                     do_bounds()

    principal global variables:
        on.hydro     called if TRUE

    principal local variables:
        termi,j      P/rho^2 for particle i, j
        av_ij        artificial viscosity ij
        termij       termi + termj + av_ij
        multi,j      additive term to acceleration of particle i, j
        term1,2     linear and quadratic terms in Noh conductivity
```

description:

This is the controlling routine to compute the hydro forces and heating. See do_hydro() and hydro_wall().

```
hydro_wall()
------------
    module:         hydro.o

    type:           void

    arguments:
        i           particle number
        bd_coord    boundary coordinate

    calls:          vect3 routines
                    kernel()
                    art_visc()

    principal global variables:
        on.hydro    called if TRUE
```

description:

This routine computes the interaction between a particle and a wall.

```
I/O MODULE:  sph_io.o
---------------------
    functions:
                    file_init()
                    write_step()
                    read_dump()
                    write_param()
                    read_param()
                    write_part()

    type:           void
```

```
   arguments:
       appnd        append to existing file if TRUE
       file         file name for write
       hist         write history if TRUE
       npart        if != 0 write particle npart only


   calls:           plot_mac  (external shell file)
```

description:

These are the i/o routines for the SPHC package. They come in three flavors: 1) write and read all problem parameters, 2) write and read all particle variable values, and 3) write a summary of a target particle to the screen. Most calls to these routines come from run_check().

```
KERNEL MODULE:  kernel.o
------------------------
   routines:        ker_setup()
                    kernel()
                    norm()

   type:            void

   arguments:
       rdiff        vector from kernel center to test point
       h            smoothing length
       W            kernel value (returned)
       del_W        gradient( W )  (vector, returned)

   calls:           vector routines

   principal global variables:
       Ker.name, "W4"          identifier for titles
       Ker.opt_space = 1.;     optimum spacing in units of h
       Ker.cutoff = 2.;        zero value radius in units of h

   principal local variables:
       nn           dimension dependent normalization factor
```

```
    z            r / h
    dsize        length of gradient vector
```

description:

This is the kernel function. Ker_setup() sets the kernel characteristics and name, kernel() returns the value and gradient of the kernel, given the smoothing length and the radius difference vector. Various kernels are available. The analytic computation should be replace by a table lookup at some future time for efficiency.

references:

See Monaghan, 1985, J. Comp. Phys. 60, 253.

```
LASER MODULE:  laser.o
----------------------
    functions:      laser_setup()
                    laser_spot()
                    laser_dep()
                    laser_pulse()
                    bremk()

    type:           double:  spot_prof()
                    others:  void

    arguments:  for bremk()
        ptr         pointer to current particle
        kb          pointer to bremsstrahlung abs coefficient
                    (returned)

    calls:          each other
                    nfel()

    principal global variables:
        on.laser        laser flag
        E_laser         total laser energy (erg)
        Fwhm            full width half-maximum (s)
        Spot_rad        laser spot radius (cm)
        Wavelength      laser wavelength (cm)
```

`F_number`	`laser f-number`
`Bremfac`	`multiplier for bremsstrahlung (def=0.5)`
`NfeO`	`preionization fraction at 1 cm from target`
`laser_pwr`	`laser power (erg/s)`
`laser_energy`	`laser energy / timestep (erg)`
`laser_etot`	`total deposited energy to date`
`laser_utot`	`total underdense deposited energy to date`

description:

This is the laser deposition module. Spot_prof() defines the profile of energy across the laser spot, laser_pulse() defines the temporal pulse shape of the laser (currently linear symmetric), laser_setup() generates the menu to set the various global parameters, bremk() computes the bremsstrahlung absorption coefficient, and laser_dep() does the deposition.

The laser deposition works this way: a linked-list of particles in the laser beam is first compiled, sorted on the X coordinate. In the overdense region, shadowed particles are dropped from the list, leaving only the critical surface particles. In the underdense region energy is deposited according to the fractional area of the beam intercepted, and the local intensity of the laser light, assuming only slight deposition in this region. The energy not deposited in the underdense region is distributed among the critical surface particles by mass. This prevents thermal numerical instabilities, and promotes uniform heating at the critical surface.

references:

Johnston and Dawson, 1973, Phys. Fl. 16, 722 (brem). User Guide, Technical Guide.

```
miss()
------

SEE:                NEIGHBOR MODULES

octree()
--------

SEE:                NEIGHBOR MODULES
```

```
NEIGHBOR MODULES
----------------
    module:         miss.o
                    octree.o

    functions       get_nbr_list()
                    init_nbr()
                    cleanup_nbr()
                    module entry function
                    utility routines
```

description:

These modules locate the particle neighbors. The particular technique is chosen at link time according to the directions in Makefile. The physics routines cal the "nbr" functions to initialize, get the neighbors, and restore the routines. The two currently implemented are "miss": a straight N^2 algorithm (looks at all other particles), and "octree", an NlogN tree scheme.

See get_nbr_list() entry for details.

```
nfel()
------
    module:         sph_utl.o

    type:           double

    arguments:
        t           temperature (K)
        z           atomic charge

    principal local variables:
        nfe         number of free electrons per ion (returned)
```

description:

Compute the number of free electrons per ion as a function of the temperature and atomic charge of a species. Currently, a very simple smooth fit is used, linear in the logs. Note that this function is used by physics routines, but is

NOT used by eos() to compute the ionization energy (eos uses its own model). This all needs to be made consistent when accurate tables are implemented.

```
reflect_particle()
------------------

   module:        sph_bounds()

   type:          int

   arguments:
      part_num - number of particle to reflect
      bdry_num - boundary to reflect about
```

description:

Reflect a particle's position and velocity with respect to a given boundary.

```
rho_compute()
-------------
   module:        rho.o

   type:          void

   arguments:     none

   calls:         vect3 routines
                  get_nbr_list()
                  do_rho()
                  do_bounds()
```

description:

Compute the density for all of the particles. Apply geometrical corrections if needed. Add the contribution due to the walls, if any.

```
rho_wall
--------
    module:         rho.o

    type:           void

    arguments:
        i           particle number
        bd_num      boundary number

    calls:          vect3 routines
                    kernel()
```

description:

Compute the wall contribution to each particle's density.

```
rhs()
-----
    module:         sphc.o

    type:           void

    arguments:      none

    calls:          vzero()
                    rho_compute()
                    eos_compute()
                    sph_bounds()
                    gravity()
                    hydro()
                    e_diff()
                    thm_diff()
                    laser_dep()
                    burn_yield()
                    poly()
```

description:

This routine is intended to do all the computations to complete the right-hand-sides of the equations (accel, dedt...). It includes all of the physics computations. It is called twice per timestep by sphc(), once for each Runge-Kutta sub-step.

```
run_check()
-----------
    module:         run_check.o

    type:           void

    arguments:      none

    calls:          sph_setup()
                    write_part()
                    write_step()
                    finish()
```

description:

This routine is the interrupt handler for SPHC. Called once per timestep, it checks to see if any keyboard input is waiting to be processed. In the following cases, action is taken:

```
'?'  -  help
'p'  -  pause for input or restart
'd'  -  debug summary this step
'w'  -  write to file this step
```

In addition, it checks for time and step number dump conditions and executes them if appropriate. Finally, it checks for "run over" status and exits if found.

The following codes are returned:

0 - error
1 - normal
2 - keyboard interrupt

3 - run finished, normal
4 - run finished, error
5 - end of relaxation

```
smooth_temp()
-------------
    module:        sph_utl.o

    type:          void

    arguments:
        len        desired smoothing distance in units of h

    calls:         init_nbr()
                   vect3 routines
                   get_nbr_list()
                   kernel()
                   cleanup_nbr()
```

description:

Smooth the initial temperature distribution. Uses the standard kernel, but through the formula

$$<Ti> = \sum_j (Tj * Wij) / \sum_j (Tj)$$

rather than normalizing by mj/rhoj under the summation. The argument len is used simply as a multiplier on the smoothing length in case a smooth sequence between no smoothing and full smoothing is desired.

```
sph_bc
------
    module:        sph_init.o

    type:          void

    arguments:
        coord            =   position of particle
        h                =   smoothing length of particle
    return:
        *numb           =   number of boundaries within 2h
                            must be <= 5
        bdry[1->numb]   =   location of closest point on each boundary
                            boundary number
                            flag if particle has crossed boundary

    calls:  none
```

description:

This user supplied routine returns information as to which particles are near the boundaries, which boundaries they are near, and whether any particles have crossed a boundary. In addition, the closest point ON each boundary is returned.

```
sph_default()
-------------
    module:         sph_default.o

    type:           void

    arguments:      none

    calls:          ker_setup()
                    sph_init0()
                    laser_setup()
```

description:

This routine sets all of the default values for variables and parameters. It also calls any default routines that are local to individual packages. It then prints the problem title to the screen.

```
sph_random()
------------
    module:         sph_utl.o

    type:           void
    arguments:      none

    calls:          system:  rnd()

    principal local variables:
        h           smoothing length
```

description:

If selected in sph_setup(), this routine adds a small random perturbation to the position of each particle. Usually, a 5–10 percent random component will prevent unnatural symmetries in the particle positions, and allow the "grid" to more easily adapt.

```
sph_setup()
-----------
    module:         sph_setup.o

    type:           void

    arguments:      none

    calls:          param routines (menu)
                    read_param()
                    system:
                        calloc(), alloc_segment(), set_extra_segment()
                    read_dump()
                    eos_init()
                    sph_init()
                    sph_random()
                    file_init()
```

description:

This is the main problem setup routine. It obtains the setup information from a series of menus and/or data file reads (if restart files are selected). It then either calls sph_init() or reads the main restart file for particle initialization. It initializes the files for history and future restart dumps. Faunal, it prints the initial timestep header to the screen.

```
sphc()
------
    module:          sphc.o

    type:            main

    arguments:       r  run from restart dump
                     p  read restart params, then execute setup routines

    calls:           sph_default()
                     sph_setup()

                     init_nbr()
                     rhs()
                     h_compute()
                     e_compute()
                     dt_compute()
                     run_check()
                     start_step()
                     laser_dep()
                     step1()
                     step2()
                     sph_bc()
                     cleanup_nbr()

    principal global variables:
        see sph.h
```

description:

This is the "main" for the sphc package. It handles the command-line parameters, call the setup routines, and supervises the overall timestep-loop computations.

```
STEP MODULE:  step.o
--------------------
    routines:        start_step()
                     step1()
                     step2()
    type:           void

    arguments:      none

    calls:          vect3 routines
                    write_part()

    principal global variables:
        struct old_part     previous timestep values
        retry               true if step is a retry
        on.relax
        eq.relax            see "run modes" in User Guide
        e_time              elapsed time
        dt                  timestep (sec)

    principal local variables:
        F11, etc.           constants needed in RK12 scheme
```

description:

These routines implement the RK12 Runge-Kutta integration scheme. Call start_step() to initialize the timestep, step1() to advance the first half-step, and step2 to initialize the final half timestep. Any additional equations need to be added to the loops in the obvious fashion. Conditional tests in these routines also implement the relaxation options.

references:

Fehlberg: NASA report # TR-R-315.

```
thermo()
--------
    module:         sph.o

    type:           void

    arguments:      none

    calls:
```

```
                    rho_compute()
                    eos_compute()
```

description:

This routine "fills in" the density, pressure, temperature, and other derived variables from the basic state variables, r, v, h, and energy.

```
thm_diff()
----------
    module:         thm_diff.o

    type:           void

    arguments:      none

    calls:          get_nbr_list()
                    vect3 routines
                    kernel()
```

description:

Compute the thermal diffusion term in the energy equation, given the thermal conduction coefficient (computed elsewhere). Currently, this routine is called if either the Noh shock-following method, electron thermal diffusion, or radiation diffusion is turned on. The equation is de/dt = (1/rho) div (thm_con grad e) = -(1/rho) div (thm flux). Note that the derivatives are in terms of the energy rather than the temperature. The physics routines must add their contribution to the particle variable old_part[].thm_con, this routine then computes the flux and the overall contribution to de/dt.

```
VECTOR ROUTINES - vect3.o
-------------------------
```

description:

This is the core vector operations module. The individual routines are listed below, along with a brief explanation. All vectors are 3-dimensional, as defined in sph.h, and are passed are pointers. The variable "sc" is a scalar, type double.

```
/*  vect3.c  --  3D vector functions  */
vzero( v1 )    /*------------------------------------ zero out a vector  */
veq( v1, v2 )   /*----------------------------------------- equate v1 = v2  */
vsum( vs, v1, v2 )     /*----------------------------------- vs = v1 + v2  */
vdiff( vd, v1, v2 )     /*------------------------------------ vd = v1 - v2  */
vav( vd, v1, v2 )       /*------------------------------- vd = (v1 + v2)/2  */
vmult( v1, sc, v2 )    /*------------------------------------ v1 = sc * v2  */
vlin( v1, sc, v2 )      /*------------------------------ v1 = v1 + sc * v2  */
vlin2( v1, v0, sc, v2 ) /*------------------------------ v1 = v0 + sc * v2  */
double dot_prod( v1, v2 )  /*--------------- return dot product of v1, v2  */
cross_prod( vp, v1, v2 )   /*------------------------------- vp = v1 x v2  */
double vect_len( v )    /*-------------------- compute length of vector v  */
double vect_len2( v )   /*----------- compute square of length of vector v  */

/*  the following routines are needed for extra seg computations only  */
x_veq( xv1, v2 )   /*-------extra seg version------------ equate xv1 = v2  */
x2_veq( v1, xv2 )   /*-------extra seg version------------ equate v1 = xv2  */
x_vlin( v1, sc, xv2 )  /*------extra seg version-------- v1 = v1 + sc * xv2  */
x_vdiff( vd, xv1, v2 )  /*------extra seg version------------ vd = xv1 - v2  */
```