

# GRAPE-6A Software Guide

Toshiyuki Fukushige

Department of General Systems Studies,  
College of Arts and Sciences,  
University of Tokyo,  
3-8-1 Komaba, Meguro-Ku, Tokyo 153-8902, Japan  
Phone : +81-3-5454-6611  
Fax : +81-3-3465-3925  
E-mail : fukushig@providence.c.u-tokyo.ac.jp

Version 1.1: November 30, 2004

Version 1.0: October 1, 2004

Beta Version: October 22, 2003

## Abstract

I give the full description of the GRAPE-6A (its commercial name is Baby GRAPE or Micro GRAPE) interface software package.

# Contents

<b>1</b>	<b>Changes</b>	<b>4</b>
1.1	Version 1.1 . . . . .	4
1.2	Version 1.0 . . . . .	4
1.3	Beta Version . . . . .	4
<b>2</b>	<b>TODO</b>	<b>4</b>
<b>3</b>	<b>To GRAPE-6 User</b>	<b>5</b>
<b>4</b>	<b>To GRAPE-5 User</b>	<b>5</b>
<b>5</b>	<b>Overview of GRAPE-6A and its operating principles</b>	<b>6</b>
<b>6</b>	<b>Linking</b>	<b>6</b>
<b>7</b>	<b>Limitations</b>	<b>7</b>
<b>8</b>	<b>Environment</b>	<b>7</b>
<b>9</b>	<b>Reference Manual for Subroutines</b>	<b>8</b>
9.1	Overview . . . . .	8
9.2	Initialization . . . . .	9
9.2.1	g6_open . . . . .	9
9.2.2	g6_close . . . . .	9
9.3	Scaling . . . . .	9
9.3.1	g6_set_tunit . . . . .	9
9.3.2	g6_set_xunit . . . . .	9
9.3.3	g6_set_overflow_flag_test_mode . . . . .	10
9.3.4	g6_get_overflow_flag_test_mode . . . . .	10
9.4	Sending data to memory . . . . .	10
9.4.1	g6_set_j_particle . . . . .	10
9.4.2	g6_set_j_particle_monly . . . . .	11
9.5	Setting current time for individual timestep algorithm . . . . .	12
9.5.1	g6_set_ti . . . . .	12
9.6	Force calculation . . . . .	12
9.6.1	g6calc_firsthalf . . . . .	12
9.6.2	g6calc_lasthalf . . . . .	12
9.6.3	g6calc_lasthalf2 . . . . .	13
9.7	Neighbour list . . . . .	13
9.7.1	g6_read_neighbour_list . . . . .	13
9.7.2	g6_get_neighbour_list . . . . .	14
9.7.3	g6_set_neighbour_list_sort_mode . . . . .	14
9.7.4	g6_get_neighbour_list_sort_mode . . . . .	15
9.8	Low level functions . . . . .	15
9.8.1	g6_reset . . . . .	15
9.8.2	g6_npipes . . . . .	15
9.8.3	g6_set_nip . . . . .	15

9.8.4	g6_set_njp . . . . .	15
9.8.5	g6_set_i_particle_scales_from_real_value . . . . .	16
9.8.6	g6_set_i_particle . . . . .	16
9.8.7	g6_get_force . . . . .	17
9.8.8	g6_get_force_etc . . . . .	17
<b>10</b>	<b>Sample programs</b>	<b>18</b>
10.1	Shared timestep . . . . .	18
10.2	Individual(Blocked) Timestep . . . . .	20
<b>11</b>	<b>Timesharing</b>	<b>23</b>

# 1 Changes

## 1.1 Version 1.1

- November 30, 2004 — kernel 2.6.5 (x86\_64, Fedora Core 2) support, fdot accuracy revised, a bug fixed in the recalculation procedure of `g6calc_lasthalf`.

## 1.2 Version 1.0

- October 1, 2004 — released

## 1.3 Beta Version

- October 14, 2003 — created

# 2 TODO

- P3M cutoff function support
- etc..

### 3 To GRAPE-6 User

The API of the GRAPE-6A software library is designed so as to be same as that of the GRAPE-6 software library (version 0.71a). Therefore, you could use programs written for GRAPE-6 by replacing the linked library (`-lg6a`).

However, there are some exceptions and notices:

(1) At present, the GRAPE-6A software library doesn't include some low-level functions:

- `g6_adjust_ip_scales`
- `g6_gueestimate_acc_etc`
- `g6_set_calculate_accel_scaling_mode`

and some high-level functions:

- `calculate_accel_by_grape6_separate_trial_noopen`
- `calculate_accel_by_grape6_noopen`

(2) the following functions do nothing in the GRAPE-6A software library:

- `g6_reinitialize`
- `g6_initialize_jp_buffer`
- `g6_flush_jp_buffer`
- `g6_reset_fofpga`

(3) In the GRAPE-6A hardware any intermittent error doesn't occur usually, unlike in the GRAPE-6 hardware. Therefore, the GRAPE-6A software library doesn't suppose any error recovery process.

### 4 To GRAPE-5 User

I prepare a software library (`-lg65`) whose API is same as that of GRAPE-5. If you write program in C language and you don't use functions for the neighbor list, you could use programs written for GRAPE-5. Some tunings to reduce the communication amount are done for this library. Fortran interface is not supported at present. It is difficult to make common API for neighbor list because GRAPE-6A needs another information (particle index) for constructing the neighbor list.

One exception is that you should send positions and masses of j-particles simultaneously using the function `g5_set_xmj` instead of `g5_set_xj` and `g5_set_mj`. The function `g5_set_xmj` is included in the GRAPE-5 software library after version 1.8.

## 5 Overview of GRAPE-6A and its operating principles

GRAPE-6A (its commercial name: Baby GRAPE or Micro GRAPE) is the smallest configuration of GRAPE-6. GRAPE-6A is a 32bit/33MHz PCI card that contains four GRAPE-6 processor chips. GRAPE-6 is the successor of GRAPE-4, designed for high-accuracy integration of gravitational N-body system using the individual timestep and Hermite scheme. It works as a backend processor, connected to a host computer through PCI.

GRAPE-6A calculates the forces and their time derivative on 48 particles, from all particles loaded into its memory. To be more precise, GRAPE-6A calculates the following

$$\mathbf{a}_i = \sum_j Gm_j \frac{\mathbf{r}_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}} \quad (1)$$

$$\dot{\mathbf{a}}_i = \sum_j Gm_j \left[ \frac{\mathbf{v}_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}} - \frac{3(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})\mathbf{r}_{ij}}{(r_{ij}^2 + \epsilon^2)^{5/2}} \right], \quad (2)$$

$$\phi_i = \sum_j Gm_j \frac{1}{(r_{ij}^2 + \epsilon^2)^{1/2}}, \quad (3)$$

where

$$\mathbf{r}_{ij} = \mathbf{x}_{p,j} - \mathbf{x}_i, \quad (4)$$

$$\mathbf{v}_{ij} = \mathbf{v}_{p,j} - \mathbf{v}_i. \quad (5)$$

Here,  $\mathbf{x}_i$ ,  $\mathbf{v}_i$ ,  $\mathbf{a}_i$ ,  $\dot{\mathbf{a}}_i$  are the position, velocity, acceleration, time derivative of acceleration of particle  $i$ ,  $G$  is the gravitational constant and  $m_j$  is the mass of particle  $j$ . With GRAPE-6A,  $G$  is fixed to unity. The parameter  $\epsilon$  is the usual plummer softening parameter.

The position and velocity of particle  $j$  have additional suffix  $p$  to denote they are “predicted” values at time  $t$  using the following formula:

$$\Delta t = t - t_j \quad (6)$$

$$\mathbf{x}_p = \frac{\Delta t^4}{24} \mathbf{a}_0^{(2)} + \frac{\Delta t^3}{6} \dot{\mathbf{a}}_0 + \frac{\Delta t^2}{2} \mathbf{a}_0 + \Delta t \mathbf{v}_0 + \mathbf{x}_0 \quad (7)$$

$$\mathbf{v}_p = \frac{\Delta t^3}{6} \mathbf{a}_0^{(2)} + \frac{\Delta t^2}{2} \dot{\mathbf{a}}_0 + \Delta t \mathbf{a}_0 + \mathbf{v}_0, \quad (8)$$

Here, we dropped the subscript  $j$  for clarity. Position  $\mathbf{x}_p$  and velocity  $\mathbf{v}_p$  are the predicted values, at time  $t$ ,  $\mathbf{x}_0$ ,  $\mathbf{v}_0$ ,  $\mathbf{a}_0$ ,  $\dot{\mathbf{a}}_0$ ,  $\mathbf{a}_0^{(2)}$  are true values of position etc at time  $t_j$ . The internal structure of GRAPE-6A looks like that in figure 1.

If you have multiple GRAPE-6A connected to a single host, they work just completely independently. In all library functions that actually communicate with GRAPE-6A hardware, you simply specify the identity of the cluster as an argument.

## 6 Linking

Following is an example to link the software library (`libg6a.a`):

```
cc prog.c -L/usr/g6a/lib -lg6a -lm
```

where I assume `/usr/g6a/lib` is the directory at which the library stored.

Following is an example to link the GRAPE-5 compatible library (`libg65.a`):

```
cc prog.c -L/usr/g6a/lib -lg65 -lm
```

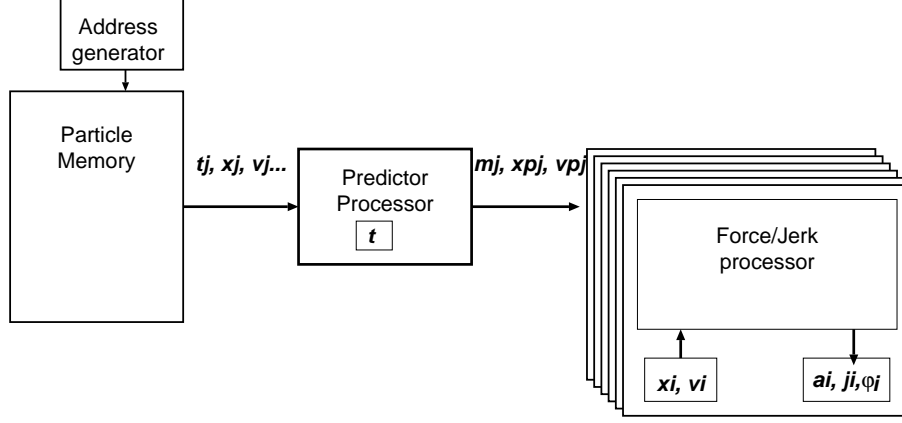


Figure 1: GRAPE-6A from the application viewpoint

## 7 Limitations

Maximum number of particles that can be stored is 65536. (Some commercial version can store up to 131072.)

## 8 Environment

At present, it is confirmed that this software library can be compiled and sample programs operates at:

- IA-32(P4/2.8G, E7205 or 865G) + Linux 2.4.7(Redhat 7.2)+ icc 5.0.1 or gcc 2.96
- IA-32(P4/2.8G, E7205 or 865G) + Linux 2.4.20-8(Redhat 9.0)+ gcc 3.2.2
- X86\_64(A64/+3500, K8T800) + Linux 2.6.5-1(Fedora Core 2)+ gcc 3.3.3

## 9 Reference Manual for Subroutines

All subroutines are written in C language. However, for simple and unified treatment of Fortran and C application programs, most of routines are written in the form which is directly callable from both Fortran and C.

### 9.1 Overview

Initialization routines

- `g6_open`: acquires the GRAPE-6A hardware.
- `g6_close`: releases the GRAPE-6A hardware.

Scaling

- `g6_set_tunit`: Specifies the binary point for time.
- `g6_set_xunit`: Specifies the binary point for position.
- `g6_set_overflow_flag_test_mode`: Specifies the test mode for result overflow
- `g6_get_overflow_flag_test_mode`: Inquires the current test mode for result overflow

Sending data to memory

- `g6_set_j_particle`: sends one  $j$ -particle data to memory.
- `g6_set_j_particle_mxonly`: sends one  $j$ -particle data (position and mass only) to memory.

Setting current time for individual timestep algorithm

- `g6_set_ti`: sets the current time for prediction.

Force calculation

- `g6calc_firsthalf`: sets scales for  $i$  particles, sends  $i$  particles, sets the numbers of  $i$ - and  $j$ -particles, and starts the calculation on GRAPE-6A.
- `g6calc_lasthalf`: waits GRAPE-6A to finish calculation and receives the results.
- `g6calc_lasthalf2`: similar to the above but get the indices for the nearest neighbor as well.

Neighbor list

- `g6_read_neighbour_list`: stores the content of the hardware neighbor list to host memory.
- `g6_get_neighbour_list`: retrieves the neighbour list for one particle.
- `g6_set_neighbour_list_sort_mode`: set the sort flag for the neighbour list.
- `g6_get_neighbour_list_sort_mode`: returns the sort mode.



## 9.2 Initialization

### 9.2.1 g6\_open

```
int g6_open(int clusterid)
```

```
subroutine g6_open(clusterid)
integer clusterid
```

Initializes the GRAPE-6A hardware and interface package for the specified cluster. The argument `clusterid` is numbered from zero.

### 9.2.2 g6\_close

```
int g6_close(int clusterid)
```

```
subroutine g6_close(clusterid)
```

This function releases the specified GRAPE-6A cluster and allows other users to acquire it.

## 9.3 Scaling

### 9.3.1 g6\_set\_tunit

```
void g6_set_tunit(int newtunit)
```

```
subroutine g6_set_tunit(newtunit)
integer newtunit
```

GRAPE-6A handles the time for predictor in 64 bit fixed point format. Thus, one need to specify where to put the binary point. The argument `newtunit` gives the point of the binary point counted from LSB. Thus, the value zero means the time expressed in GRAPE-6A is truncated to integral values, while, say, 50 means the resolution is  $2^{-50}$ . The default value is 51.

What value should be used depends on the system of units that the application uses. If one uses the “standard” or so-called Heggie units, the default value should be fine, but one must be careful that the time would not overflow (there are only 13 bits available above the binary point). If the simulation covers the time much longer than  $10^3$ , the default should be changed.

If you are using GRAPE for, say, galactic simulation and use real physical unit such as Myr as the time unit, you might want to use smaller value for `newtunit` than the default, since the default does not cover the Hubble time, and the resolution is too high (around 1 milliseconds).

When time unit is changed through the call to this function, the content of the GRAPE-6A memory becomes inconsistent with the time unit, and if individual timestep is used, all “*j*-particles” should be resent.

### 9.3.2 g6\_set\_xunit

```
void g6_set_xunit(int newxunit)
```

```
subroutine g6_set_xunit(newxunit)
integer newxunit
```

Similar to `g6_set_tunit`, but gives the binary point for space coordinate. The default is again 51, which is good if the size of the system is order unity. If your system is much larger or much smaller, you should change the default value accordingly.

### 9.3.3 `g6_set_overflow_flag_test_mode`

```
void g6_set_overflow_flag_test_mode(int force_test_mode,
    int jerk_test_mode, int pot_test_mode)
```

```
subroutine g6_set_overflow_flag_test_mode(force_test_mode,
    jerk_test_mode, pot_test_mode)
integer force_test_mode, jerk_test_mode, pot_test_mode
```

This function specifies which overflow flags are checked by the library. To let the library test the overflow for force, you set `force_test_mode` to one, and to ignore overflow, set it to zero.

The most likely use of this function is to set `jerk_test_mode` to zero if you do not use it for time integration (for example, if you use leapfrog).

### 9.3.4 `g6_get_overflow_flag_test_mode`

```
void g6_get_overflow_flag_test_mode(int *force_test_mode,
    int *jerk_test_mode, int *pot_test_mode)
```

```
subroutine g6_get_overflow_flag_test_mode(force_test_mode,
    jerk_test_mode, pot_test_mode)
integer force_test_mode, jerk_test_mode, pot_test_mode
```

This function returns the current values of the flags to test overflow. Initially all three flags are set to one.

## 9.4 Sending data to memory

### 9.4.1 `g6_set_j_particle`

```
int g6_set_j_particle(int clusterid,
    int address,
    int index,
    double tj, /* particle time */
    double dtj, /* particle time */
    double mass,
    double a2by18[3], /* a2dot divided by 18 */
    double a1by6[3], /* a1dot divided by 6 */
    double aby2[3], /* a divided by 2 */
    double v[3], /* velocity */
    double x[3] /* position */)
```

```
integer function g6_set_j_particle(clusterid, address, index, tj, dtj, mass,
    a2by18, a1by6, aby2, v, double x)
integer clusterid, address, index
double precision tj, dtj, mass, a2by18(3), a1by6(3), aby2(3), v(3), x(3)
```

This function sends the so-called *j*-particle data to the memory unit of GRAPE-6A. `x`, `v`, `aby2`, `a1by6`, `a2by18` are position, velocity, half of the acceleration, 1/6 of the first time derivative of the acceleration, and 1/18 of the second time derivative. These must all be the arrays of size three. `tj`, `dtj` and `mass` are the time, timestep and mass of the particle. `address` is the location in the GRAPE-6A memory to store the particle, starting from index 0, and `index` is the identifier for the particle itself, which may be different from `address` above.

The time and timestep must be acceptable for the blockstep algorithm, which means that the timestep must be the powers of two (negative values allowed, but smaller than the time resolution set by `g6_set_tunit` should result in an error). Also the time must be an exact integer multiple of the timestep.

Note that the GRAPE-6A predictor unit can make use of the second time derivative of the force, unlike the GRAPE-4 predictor unit which can use only up to the first time derivative. If the application program cannot provide the second time derivative, it must give the pointer of the array with size three filled with zeros.

The parameter `index` is introduced to GRAPE-6A to achieve

- Inhibition of the self-interaction base on particle identity
- Easier use of the neighbor list than GRAPE-3/4/5

The function `g6calc_firsthalf` also has this index as an argument. Thus, if the indices are the same, the accumulation of the result is skipped on hardware. Thus, effectively, one can achieve something like the following:

```
C calculate the force on particle i from all other particles
  do j = 1, n
    if (index(i) .ne. index(j)) then
C      do the actual force calculation
    endif
  enddo
```

The functions for the neighbor list will return the list of particles using this index, and not the location of the particles in the memory as used to be so on older GRAPE systems. This should make the application program somewhat simpler, in particular when the number of particles is larger than what fits on the memory of GRAPE-6A.

#### 9.4.2 `g6_set_j_particle_mxonly`

```
int g6_set_j_particle_mxonly(int clusterid,
                             int address,
                             int index,
                             double mass,
                             double x[3] /* position */)

integer function g6_set_j_particle_mxonly(clusterid, address, index, mass,x)
integer clusterid, address, index
double precision mass, x(3)
```

This function works in the same way as `g6_set_j_particle` except that it sets only mass and position. This is handy if the application does not uses the predictor.

## 9.5 Setting current time for individual timestep algorithm

### 9.5.1 g6\_set\_ti

```
void g6_set_ti(int clusterid, double ti)
```

```
subroutine g6_set_ti_(clusterid, ti)
integer clusterid
double precision ti
```

This function sets the present time ( $t_i$ ) for the predictor unit.

## 9.6 Force calculation

### 9.6.1 g6calc\_firsthalf

```
void g6calc_firsthalf(int clusterid,
                     int nj,
                     int ni,
                     int index[],
                     double xi[][3],
                     double vi[][3],
                     double fold[][3],
                     double jold[][3],
                     double phiold[],
                     double eps2,
                     double h2[])

subroutine g6calc_firsthalf(clusterid, nj, ni, index, xi, vi, fold,
                           jold, phiold, eps2, h2)
integer clusterid, nj, ni, index(*),
double precision xi(3,*), vi(3,*), fold(3,*), jold(3,*), phiold(*),
               eps2, h2(*)
```

This function just calls `g6_set_ip_scales`, `g6_set_i_particle`, `g6_set_nip` and `g6_set_njp` in that order. In other words, it sets the necessary scalings for  $i$ -particles, sends them to GRAPE-6, set number of particles (both  $i$  and  $j$ ) and starts the calculation. Note that `fold`, `jold` and `phiold` are used to determine the scaling. Therefore, they should have the value corresponding to the particle.

The application program can calculate the force by calling this function and `g6calc_lasthalf` in that order.

### 9.6.2 g6calc\_lasthalf

```
int g6calc_lasthalf(int clusterid,
                   int nj,
                   int ni,
                   int index[],
                   double xi[][3],
                   double vi[][3],
                   double eps2,
```

```

        double h2[],
        double acc[][3],
        double jerk[][3],
        double pot[])

integer function  g6calc_lasthalf(clusterid, ni, nj, index, xi, vi,
        eps2, h2, acc, jerk, pot)
integer clusterid, nj, ni, index
double precision xi(3,*), vi(3,*), eps2, h2(*), acc(3,*), jerk(3,*), pot(*)

```

This function waits for the end of calculation and retrieves the calculated result. In the case of the scaling error, this function tries to adjust the scaling factors appropriately and retries the calculation. If some hardware error occurs and is detected, this function returns non-zero value. Otherwise the return value is zero.

### 9.6.3 g6calc\_lasthalf2

```

int g6calc_lasthalf2(int clusterid,
        int nj,
        int ni,
        int index[],
        double xi[][3],
        double vi[][3],
        double eps2,
        double h2[],
        double acc[][3],
        double jerk[][3],
        double pot[],
        int nnbindex[])

integer function  g6calc_lasthalf2(clusterid, ni, nj, index, xi, vi,
        eps2, h2, acc, jerk, pot, nnbindex)
integer clusterid, nj, ni, index, nnbindex(*)
double precision xi(3,*), vi(3,*), eps2, h2(*), acc(3,*), jerk(3,*), pot(*)

```

Same as `g6calc_lasthalf` except that this function returns `nnbindex`, the indices for the nearest neighbors.

## 9.7 Neighbour list

### 9.7.1 g6\_read\_neighbour\_list

```

int g6_read_neighbour_list(int clusterid)

integer function g6_read_neighbour_list(clusterid)
integer clusterid

```

This function transfers the content of the hardware neighbor list of GRAPE-6A to the working memory of the interface library. Return value of zero means successful completion, -1 some internal

error and 1 overflow of the hardware neighbour list memory. This function must be called only once after `g6_calc` (or `g6calc_lasthalf`) is called.

Zero return value means normal end. Negative values indicate some hardware error (you need to reset the hardware and restart calculation). Positive return values mean overflow (radius too large).

Note that GRAPE-6A always calculates the force, and therefore neighbour lists, for 48 particles, even when you send only one particle with `g6calc_firsthalf`. For particles not supplied by the present call, the values last set are used. Thus, if you have neighbour list overflow with force calculation for less than 48 particles, that may be caused by the values of the neighbour sphere radius for “unused” locations.

If you want to use the neighbour list with less than 48 particles, always make sure that you do not set large values to “unused” locations. The simplest way to guarantee this is of course just to copy the last particle to all “unused” locations.

### 9.7.2 `g6_get_neighbour_list`

```
int  g6_get_neighbour_list(int clusterid,
                           int  ipipe,
                           int  maxlength,
                           int * nblen,
                           int  nbl[])
```

```
integer function  g6_get_neighbour_list(clusterid, ipipe, maxlength, nblen, nbl)
integer clusterid ipipe maxlength nblen, nbl(*)
```

This function extracts the actual neighbor list of particle calculated for particle `ipipe` from the working memory of the interface library prepared by `g6_read_neighbour_list`. The argument `maxlength` gives the size of the array `nbl`. Return value of zero means successful completion and 1 overflow. On successful return, `nblen` has the length of the neighbour list and `nbl` actual list (sorted by the indices).

### 9.7.3 `g6_set_neighbour_list_sort_mode`

```
int  g6_set_neighbour_list_sort_mode(int mode)
                           int  ipipe,
                           int  maxlength,
                           int * nblen,
                           int  nbl[])
```

```
integer function  g6_set_neighbour_list_sort_mode(mode)
integer mode
```

This function sets the sort option flag for the neighbour list functions. When set to 1, the neighbour list returned by `g6_get_neighbour_list` is sorted by the indices of the neighbors. This sorting takes rather significant time. If you do not need your neighbour list sorted, you can get some speedup by calling this function with argument 0. Initial value of flag is 1.

#### 9.7.4 g6\_get\_neighbour\_list\_sort\_mode

```
int g6_get_neighbour_list_sort_mode(int mode)
                                   int ipipe,
                                   int maxlength,
                                   int * nblen,
                                   int nbl[])
```

integer function g6\_get\_neighbour\_list\_sort\_mode

This function returns the sort option flag for the neighbour list.

### 9.8 Low level functions

#### 9.8.1 g6\_reset

```
int g6_reset_(int *clusterid)
```

```
subroutine g6_reset(clusterid)
integer clusterid
```

Performs the hardware reset. Usually one need not use this function, except for error recovery.

#### 9.8.2 g6\_npipes

```
int g6_npipes()
```

integer function g6\_npipes

Returns the number of pipelines available on the particular GRAPE-6A system in use. In most cases, it just returns 48, the number of physical pipelines per chip.

#### 9.8.3 g6\_set\_nip

```
void g6_set_nip(int clusterid, int nip)
```

```
subroutine g6_set_nip(clusterid, nip)
integer clusterid, nip
```

Set the number of particles on which the forces (etc) are calculated. The maximum value for nip is 48, which is the number of virtual pipelines per chip. At least currently, the range of the arguments are not checked. So the application programmer has the responsibility to put them within the allowed range.

#### 9.8.4 g6\_set\_njp

```
void g6_set_njp(int clusterid, int njp)
```

```
subroutine g6_set_njp(clusterid, njp)
integer clusterid, njp
```

This function sets the number of the particles on the memory to on-chip registers of GRAPE-6A. As a side effect, it let GRAPE-6A start the calculation.

### 9.8.5 g6\_set\_ip\_scales\_from\_real\_value

```
void g6_set_ip_scales_from_real_value(int clusterid,  
                                     int address,  
                                     double acc[3],  
                                     double jerk[3],  
                                     double phi, double jfactor,  
                                     double ffactor)
```

```
subroutine g6_set_ip_scales_from_real_value(clusterid, address, acc, jerk, phi,  
                                           jfactor, ffactor)
```

```
integer clusterid, address
```

```
double precision acc(3), jerk(3), phi, jfactor, ffactor
```

This function sets the binary point for the internal accumulator for force etc. GRAPE-6A performs the accumulation of the force etc is done in fixed-point format, in order to simplify the hardware and yet extend effective accuracy. Therefore, the hardware should know where to place the binary point before starting the calculation, and in order to do so the library function should know approximate size of force (etc). A good guess is the values at the previous timestep, and therefore within the time integration routine one can just pass the actual values of acceleration, jerk and potential to this routine.

Two additional parameters, `jfactor` and `ffactor`, are both used to allow more subtle control on the scaling of jerk. While acceleration and potential are accumulated in 64-bit format, jerk is accumulated in 32 bit. Thus, in order to allow reasonable accuracy, I chose the margin for jerk much smaller than that for acceleration and potential. The parameter `jfactor` is used to change the margin. The passed values of jerk is *multiplied* by `jfactor` before being used to calculate the scale. Thus, suppling `jfactor` larger than unity significantly reduce the chance of overflow, but might result in slightly less accurate value of jerk. I do not recommend the use of `jfactor` > 10.

The parameter `ffactor` is used to calculate the scale for jerk from the acceleration. To be specific, the scaling factor is calculated by

```
jmax = fabs(jerk[k])*(*jfactor) + fabs(acc[k])*(*ffactor);
```

This parameter is used to further reduce the chance of overflow. Since the jerk is the time derivative of the acceleration, acceleration divided by the timestep would give pretty good upper limit for the jerk.

For the first call, the use of this function is a bit problematic, since the application program might not have good knowledge on how larger is the force on a particle. In this case, I'd recommend to initialize the arrays for acceleration etc with fairly large values and then do the force calculation twice. In the first call, a good guess for the actual value is obtained. However, the calculated force itself should not be used, since the binary point might be inappropriate.

### 9.8.6 g6\_set\_i\_particle

```
void g6_set_i_particle(int clusterid,  
                      int address,  
                      int index,  
                      double x[3], /* position */  
                      double v[3], /* velocity */  
                      double eps2,
```



```
double h2)
```

```
subroutine g6_set_i_particle(clusterid, address, index, x, v, eps2, h2)
integer clusterid, address, index,
double precision x(3), v(3), eps2, h2
```

This function sends the data of particles on which the force etc are to be calculated. The argument `address` is the location within GRAPE-6 register files, and it should be within the range of  $[0, 47]$ . The meaning of the index is described in the section for `g6_set_j_particle`. `eps2` is the softening parameter squared. Setting this zero would not cause error, if `index` is correctly handled, since the self-interaction is avoided through this index. `h2` is the radius of the neighbor sphere.

### 9.8.7 g6\_get\_force

```
int g6_get_force(int clusterid,
                 double acc[][3],
                 double jerk[][3],
                 double phi[],
                 int flag[])

integer function g6_get_force(clusterid, acc, jerk, phi, flag)
integer clusterid, flag(*)
double precision acc(3,*), jerk(3,*), phi(*)
```

This function returns the calculated result. If the calculation is not finished, it waits until the end.

If non-zero value is returned, it means some error and that the recalculation is required. The meaning of the non-zero error code is not specified yet. Arguments `acc`, `jerk`, `phi` are the calculated acceleration, jerk and potential, respectively. `flag` indicates the status.

### 9.8.8 g6\_get\_force\_etc

```
int g6_get_force_etc(int clusterid,
                     double acc[][3],
                     double jerk[][3],
                     double phi[],
                     int nnbindindex[],
                     int flag[])

integer function g6_get_force_etc(clusterid, acc, jerk, phi, nnbindindex, flag)
integer clusterid, flag(*), nnbindindex(*)
double precision acc(3,*), jerk(3,*), phi(*)
```

Essentially the same as `g6_get_force`, except that this function returns the indices of the nearest neighbors in `nnbindindex` as well. Note that the returned indices are NOT the memory address but the index set by `g6_set_i_particle`.

## 10 Sample programs

### 10.1 Shared timestep

The following code evaluates accelerations by  $O(N^2)$  direct summation. Actually, this sample is a part of the test program `s9` in the library package.

```
void force_grape(double x[][3], double m[],
                double eps, double a[][3],
                double pot[], int n)
{
    int i,j,k,nn,ii,npipe;
    double tj,dtj,eps2,h2,h;
    double jerk[48][3];
    int flag[48];
    double time;
    double xi[48][3],vi[48][3],h2i[48];
    double foldi[48][3],joldi[48][3],phioldi[48];
    int index[48];
    int clusterid,tunit,xunit;
    int aflag,jflag,pflag,aflag2,jflag2,pflag2;
    int maxlength,nblen,nbl[256],ret,ret2,inb,nblh[256],nnbh,nflag;

    tj = 0.0;
    dtj = 1.0;
    time = 0.0;
    eps2 = eps*eps;
    h2 = 0.3*0.3;
    maxlength = 255;

    npipe = g6_npipes();
    tunit = 51; /* 2^51 */
    g6_set_tunit(tunit);
    xunit = 50; /* 2^50 */
    g6_set_xunit(xunit);

    clusterid=0;
    g6_open(clusterid);

    aflag = 1; jflag = 0; pflag = 1;
    g6_set_overflow_flag_test_mode(aflag,jflag,pflag);
    nflag =1;
    g6_set_neighbour_list_sort_mode(nflag);

    for(i=0;i<n;i++) g6_set_j_particle_mxonly(clusterid, i, i, m[i],x[i]);

    g6_set_ti(clusterid, time);
    for(i=0;i<n;i+=npipe){
        nn = npipe;
        if(n-i<npipe) nn= n - i;
        for(ii=0;ii<nn;ii++){
            index[ii] = i+ii;
            for(k=0;k<3;k++){
```

```

        xi[ii][k] = x[i+ii][k];
        vi[ii][k] = 0.0;
        foldi[ii][k] = a[i+ii][k];
        joldi[ii][k] = 0.0;
    }
    h2i[ii] = h2;
    phioldi[ii] = pot[i+ii];
}
g6calc_firsthalf(clusterid,n,nn,index,xi,vi,foldi,joldi,phioldi,eps2,h2i);
g6calc_lasthalf(clusterid,n,nn,index,xi,vi,eps2,h2i,a[i],jerk,&pot[i]);

ret=g6_read_neighbour_list(clusterid);
if(ret==1){
    printf("NB list overflow in hardware\n");
}else{
    for(ii=0;ii<nn;ii++){
        ret2=g6_get_neighbour_list(clusterid,ii,maxlength,&nblen,nbl);
        if(ret2==1) printf("NB list > maxlength\n");
    }
}
}
g6_close(clusterid);
}

```

## 10.2 Individual(Blocked) Timestep

The following is a part of an example code for individual(blocked) timestep algorithm. Actually, this sample is a part of the test program `s8` in the library package. And, the fortran version, `s8f`, is also included in the library package.

```
clusterid=0;
tunit = 51;                      /* 2^51 */
g6_set_tunit(tunit);
xunit = 50;                      /* 2^50 */
g6_set_xunit(xunit);
g6_open(clusterid);

for(i=0;i<n;i++){
  for(k=0;k<3;k++){
    a0[i][k] = 1.0;
    adot0[i][k] = 100.0;
  }
  pot[i] = -1.0;
}
force(x0,v0,m,ti,eps,a0,adot0,pot,n); /* initial step's force */
force(x0,v0,m,ti,eps,a0,adot0,pot,n);
initial_energy(pot,x0,v0,m,n,eps,&init_ene);
initial_timestep(a0,adot0,dti,n,eta_s);
for(i=0;i<n;i++) index[i] = i;
set_particle_on_grape(x0,v0,a0,adot0,m,ti,dti,n);

nsame = n;
do{
  double nextt;
  sort_timestep_m(0,nsame-1,dti,index);
  time = ti[index[0]] + dti[index[0]];

  isame = 0;
  do{
    isame++;
    nextt = ti[index[isame]] + dti[index[isame]];
  }while((time == nextt)&&(isame<n));
  nsame = isame;

  for(i=0;i<nsame;i++){
    idi = index[i];
    predict(time,x1[idi],v1[idi],ti[idi],x0[idi],v0[idi],a0[idi],adot0[idi]);
    ti[idi] = time;
  }
  g6_set_ti(clusterid, time);

  for(i=0;i<nsame;i+= npipe){
    ni = npipe;
    if(i+ni>nsame) ni = nsame - i;

    for(ii=0;ii<ni;ii++){
      idi = index[i+ii];
      index2[ii] = index[i+ii];
    }
  }
}
```

```

    for(k=0;k<3;k++){
        xi[ii][k] = x1[idi][k];
        vi[ii][k] = v1[idi][k];
        foldi[ii][k] = a0[idi][k];
        joldi[ii][k] = adot0[idi][k];
    }
    h2i[ii] = h2;
    phioldi[ii] = pot[idi];
}

g6calc_firsthalf(clusterid,n,ni,index2,xi,vi,foldi,joldi,phioldi,eps2,h2i);
g6calc_lasthalf(clusterid,n,ni,index2,xi,vi,eps2,h2i,tmpa,tmpadot,tmpapot);

for(ii=i;ii<(ni+i);ii++){
    idi = index[ii];
    for(k=0;k<3;k++){
        a1[idi][k] = tmpa[ii-i][k];
        adot1[idi][k] = tmpadot[ii-i][k];
    }
    pot[idi] = tmpapot[ii-i];

    correct(x1[idi],v1[idi],x0[idi],v0[idi],a0[idi],adot0[idi],
            a1[idi],adot1[idi],&dti[idi],time,eta);

    for(k=0;k<3;k++){
        x0[idi][k] = x1[idi][k];
        v0[idi][k] = v1[idi][k];
        a0[idi][k] = a1[idi][k];
        adot0[idi][k] = adot1[idi][k];
    }

    for(k=0;k<3;k++){
        aby2[k] = (1.0/2.0)*a0[idi][k];
        alby6[k] = (1.0/6.0)*adot0[idi][k];
        a2by18[k] = 0.0;
    }
    g6_set_j_particle(clusterid, idi, idi, ti[idi], dti[idi], m[idi],
        a2by18, alby6, aby2, v0[idi], x0[idi]);
}

if( time >= eouttime){
    energy(time,pot,x1,v1,m,n,eps,init_ene);
    eouttime += deouttime;
}

if( time >= releasetime) {
    releasetime += dreleasetime;
    g6_close(clusterid);
    g6_open(clusterid);
    set_particle_on_grape(x0,v0,a0,adot0,m,ti,dti,n); /* send all j-particles */
}
}
}while(time<endtime);

```

```
g6_close(clusterid);
```

## 11 Timesharing

The GRAPE-6A interface offers a rather primitive method for sharing the GRAPE-6A hardware by several programs running simultaneously. A program acquires GRAPE-6A by calling `g6_open`. If someone is already using GRAPE-6A, `g6_open` prints some message and put the process into the sleep state. When the hardware is released, the process is waken up and `g6_open` returns. A program can release GRAPE-6A by calling `g6_close`. Thus, a program occupies GRAPE-6A hardware between the calls to `g6_open` and `g6_close`.

In order to attain the sharing of GRAPE-6A between multiple users, therefore, programmers must write their program so that it releases GRAPE-6A at an reasonable time interval, which is around 1 minutes.

## Acknowledgment

I am very grateful to Junichiro Makino for much effort in preparing "*GRAPE-6 User's Guide*", from which most part of this document is copied, Astushi Kawai for much effort in preparing PCI interface program, and Pawel Cieciela for help in supporting X86\_64 architecture.

## AUTHOR

Toshiyuki Fukushige